AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY
Faculty of Electronics, Communications and Automation

Otso Palonen

# OBJECT-ORIENTED IMPLEMENTATION OF OPC UA INFORMATION MODELS IN JAVA

Thesis submitted for examination for the degree of Master of Science in Technology

Espoo 29.1.2010

Thesis supervisor:

Prof. Kari Koskinen

Thesis instructor:

M.Sc.(Tech.) Jouni Aro

AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY

ABSTRACT OF THE
MASTER'S THESIS

Author: Otso Palonen

Title: Object-oriented implementation of OPC UA information models in Java

Date: 29.1.2010 Language: English Number of pages: 9+72

Faculty: Faculty of Electronics, Communications and Automation

Professorship: Information and Computer Systems in Automation Code: AS-116

Supervisor: Prof. Kari Koskinen

Instructor: M.Sc.(Tech.) Jouni Aro

OPC UA is a new specification for communication between information systems. Its major developments compared to OPC are modern transport and security options and the possibility of expressing the semantics of data in an interoperable and expandable way. This thesis is about how information models, which define the semantics of data, should be supported in OPC UA servers and how the servers can be linked to the data the models are representing.

Those parts of the OPC UA specification that are related to this thesis as well as existing implementations are presented. The reasons for using information models is applications are explained and some examples of information models are presented.

Information model support is added to a Java Sample Server, based on information models defined in XML files of a well-known format. Several models may be added to a server, and each may have its own binding to an underlying data system. The bindings are implemented using a system for adding custom code to a server, and support runtime additions to the server.

In addition, an example use case using the developed information model instantiation and code binding tools is presented.

Keywords: OPC UA, information modelling, Java

AALTO-YLIOPISTO
TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Otso Palonen

Työn nimi: OPC UA -tietomallien oliopohjainen toteutus Java-kielellä

Päivämäärä: 29.1.2010 Kieli: Englanti Sivumäärä: 9+72

Tiedekunta: Elektroniikan, tietoliikenteen ja automaation tiedekunta

Professuuri: Automaation tietotekniikka Koodi: AS-116

Valvoja: Prof. Kari Koskinen

Ohjaaja: DI Jouni Aro

OPC UA on uusi määrittely tietojärjestelmien väliseen viestintään. Sen suurimmat edistysaskeleet verrattuna OPC:hen ovat modernit siirtoprotokollat, tietoturvaominaisuudet sekä mahdollisuus datan semantiikan ilmaisemiseen laajennettavasti ja valmistajariippumattomasti. Tämä diplomityö käsittelee tietomallien, joita käytetään datan semantiikan määrittelemiseen, tukemista OPC UA -palvelimissa ja palvelimien kytkemistä mallien esittämään dataan.

OPC UA -määrittelyä käsitellään siinä määrin kuin se liittyy työn aiheeseen ja esitellään olemassaolevia toteutuksia. Työssä selvitetään syitä tietomallien käyttöön sovelluksissa ja esitellään muutamia esimerkkejä.

Tietomallituki lisätään työssä Java Sample Server -palvelimeen, perustuen hyvin määritellyssä XML-muodossa oleviin tietomalleihin. Useita malleja voidaan lisätä palvelimelle, ja jokaisella voi olla oma kytkentänsä taustajärjestelmän dataan. Kytkentöjen lisäämistä varten luodaan koodinsitomisjärjestelmä, joka tukee myös ajonaikaisia lisäyksiä palvelimeen.

Lisäksi työssä esitellään esimerkkikäyttötapaus käyttäen kehitettyjä tietomallien lisäys- ja koodinsitomistyökaluja.

Avainsanat: OPC UA, tietomallinnus, Java

# Preface

This thesis has been made at Prosys PMS Ltd, Espoo, Finland during the latter half of 2009. It has been made as a part of a Tekes-funded OPC UA development project. The work has taught me a great deal of open-minded, yet critical thinking and given me a chance to flex my intellectual muscles, and at points, exert them maximally.

I wish to thank my instructor Jouni Aro for his invaluable instruction and for feeding my enthusiasm (as well as taking the time to point out all my mistakes in a gentle fashion), and my supervisor Kari Koskinen for his support during the whole process. In addition, I would like to thank Ilkka Seilonen for reviewing the manuscript, and Pekka Aarnio for introducing me to the world of semantics. Worthy of praise are also all the people who worked in the Java project, as well as the good people at Neste Jacobs. And of course everyone at Prosys deserves a heartfelt thanks for providing such a great atmosphere to work in.

Of course, my dear mother and my two sisters as well as their families have supported me during the writing process, as they always have. For that, and everything else, I thank them. And my friends have to be thanked for keeping me (at least relatively) sane, I can only hope they keep up the good work.

This work is dedicated to the memory of my father, Vesa Palonen.

Otaniemi, 29.1.2010

Otso Palonen

# Contents

# List of Acronyms

| | |
|---|---|
| API | Application Programming Interface |
| COM | Component Object Model |
| CSV | Comma Separated Values |
| DCOM | Distributed Component Object Model |
| DCS | Distributed Control System |
| DOM | Document Object Model |
| EDDL | Electronic Device Description Language |
| ERP | Enterprise Resource Planning |
| FDT | Field Device Tool |
| GUID | Globally Unique IDentifier |
| IDE | Integrated Development Environment |
| MES | Manufacturing Execution System |
| MIMOSA | Machinery Information Management Open Systems Alliance |
| NAPCON | Neste Advanced Process CONtrol |
| OPC | OLE for Process Control |
| OPC A&E | OPC Alarms and Events |
| OPC DA | OPC Data Access |
| OPC UA | OPC Unified Architecture |
| OWL | Web Ontology Language |
| RAD | Rapid Application Development |
| RDF | Resource Description Framework |
| RPC | Remote Procedure Call |
| SAX | Simple API for XML |
| SCADA | Supervisory Control And Data Acquisition |
| SDK | Software Development Kit |
| SOA | Service Oriented Architecture |
| SOAP | Simple Object Access Protocol |
| SPARQL | SPARQL Protocol and RDF Query Language |
| TCP | Transmission Control Protocol |
| TMLDB | Task Manager Local DataBase |
| UML | Unified Modelling Language |
| URI | Uniform Resource Identifier |
| WS | Web Services |
| XML | eXtensible Markup Language |
| XSL | eXtensible Stylesheet Language |
| XSLT | XSL Transformations |

# OPC UA Terminology

| | |
|---|---|
| AddressSpace | The collection of information that an OPC UA Server makes visible to its Clients.[6] |
| Attribute | A primitive characteristic of a Node. All Attributes are defined by OPC UA, and may not be defined by Clients or Servers. Attributes are the only elements in the AddressSpace permitted to have data values.[6] |
| DataType | The DataType is used to define the data type of a Variable.[7] |
| ExpandedNodeId | An expanded version of NodeId, which may have an address space URI in addition to the index specific to a server. |
| Information Model | An organizational framework that defines, characterizes and relates information resources of a given system or set of systems.[6] |
| Method | A callable software function that is a component of an Object.[6] |
| ModellingRule | Part of a type declaration which defines how a part of that type will be used in instantiation.[7] |
| Node | The fundamental component of an AddressSpace.[6] |
| NodeId | An unique identifier (in the context of a server) for a Node. |
| NodeClass | The class of a Node in an AddressSpace.[6] The eight possible NodeClasses are further explained in section 2.3.2. |
| Object | A Node that represents a physical or abstract element of a system.[6] |
| Property | Variables that are the TargetNode for a HasProperty Reference. Properties describe the characteristics of a Node.[7] |
| Reference | An explicit relationship (a named pointer) from one Node to another.[6] |
| Service | A Client-callable operation in an OPC UA Server.[6] |
| Variable | A Variable is a Node that contains a value.[6] |

# 1 Introduction

## 1.1 Background and motivation

OPC Unified Architecture (abbreviated OPC UA) is a new specification which seeks to create a common infrastructure for sharing information between different systems. Interoperability between information systems of not just different manufacturers, but also of different purposes has long been a coveted goal. Whereas OPC, UA's predecessor, enabled data from automation instruments of different manufacturers to be accessed using one protocol, OPC UA is an attempt to create an extensible specification to expose and access data regardless of the purpose or meaning of the data, enabling interoperability between any number of systems of different purposes. As Mylopoulos and Papazoglou summarize it in the series foreword of *A Semantic Web Primer*[1], "Future information systems will have to support smooth interaction with a large variety of independent multivendor data sources and legacy applications, running on heterogenous platforms and distributed information networks. Metadata will play a crucial role in describing the contents of such data sources and in facilitating their integration." Even though they are not talking about OPC UA in specific but of semantic models at large, OPC UA is trying to achieve just those goals. To that end, UA uses some of the latest technological paradigms, such as SOA and Web Services, while still retaining compatibility with legacy systems. Building on the success of the previous generation of OPC products, OPC UA aims to become a IEC standard and be as successful as its predecessor in automation applications. In addition it is hoped that it will expand the possible applications to include new areas. Such new uses could range from communications between an embedded device and a DCS (Distributed Control System) to communication between ERP (Enterprise Resource Planning) systems, all using the same specification.

Exposing the semantics of data is very important, and as such information models are one of the cornerstones of OPC UA. They are used to represent information in a structured way. As OPC UA applications do not have rigid, built-in models for the semantic meaning of the data, new models can be defined beside and on top of previous information models to add semantics to the data. An example of this would be a valve vendor creating an information model for their valve. This model could be an extension of a more generic information model for process devices (which in turn builds on the basic OPC UA information model), and use the concepts defined therein. Creating these information models and enabling their use in applications is crucial to utilizing OPC UA to its fullest potential. Several information models have been defined, by the OPC Foundation and also by other organizations, to unify the modelling of information in specific fields. However, these models only serve as a base for vendor-specific models which are the topmost and final layer in any hierarchy of information models. Information models need to be defined to best capture the semantics of the application field, and it is a challenge to make client and server implementations which work with diverse information models.

## 1.2 Scope and objectives of the work

This thesis explores the use of information models in OPC UA server applications. The solution developed in this thesis is a part of the Finnish project to create a Java Sample Server, which is a Java counterpart to the C# Sample Server by the OPC Foundation. The solution is inteded to enable use of information models in the Java Sample Server. The work has been done simultaneously with the development of a propietary SDK for creating servers and clients on top of the Java stack. In addition, a OPC UA solution for an existing system has been under development by Prosys and Neste Jacobs. The lessons learned from these two projects have also been incorporated into this thesis.

The main research question of this thesis is what kind of issues surround adding information models to OPC UA servers. It consists of adding the types and instances into the server's address space. The requirements this places on the implementation of servers needs to be considered, as well as the question as to what good the information models actually are in the end. Since an OPC UA information model contains the semantics of its application domain, (whether a class of devices or a single valve) OPC UA applications can potentially use this information in their operation in novel ways. The possibilities this rises are beyond the scope of this thesis, but semantic models and their relation to OPC UA information models are explored briefly. A secondary, but important question is how the server's data should be bound to an underlying system, and how custom operations can be added to the server. Related areas that are also of interest in the literature review are the creation of the information models and their validation.

Adding information model handling capabilities to the Java Sample Server is implemented in this thesis, and a prototype of binding the data presented by the server to customized code is also made. Different implementation strategies for both will be considered, after which a solution will be implemented. The information model handling classes will be based on the existing solution and be integrated with it, but as the Java Sample Server had minimal support for information models, the existing solution is also modified to accommodate the new functionality. An example that uses the developed capabilities is demonstrated.

## 1.3 Research methods

The two main sources for information in this work have been literature and the existing implementations. Relatively few publications exist on this topic, and it is an area which will surely receive more attention as the possibilities of information models begin to clarify. However, the OPC UA specifications as well as Damm et al.'s *OPC Unified Architecture*[2] are core references. As for the existing implementations, their source code has been studied to figure out the specifics of each implementation. Also, lessons learned during other development work has also contributed to this work.

## 1.4 Structure of the work

The work is divided into seven chapters, the first of which is this introduction which has outlined the background and scope of this thesis. The second will give an overview of OPC UA, along with its history and relation to OPC. The third chapter presents the current implementations of OPC UA in software. The fourth introduces information models along with some examples. The fifth chapter covers the use of information models in applications, and it tries to shed light on how information models can be supported in server applications. It also presents tools for defining information models. The sixth chapter details the implementation made as part of the Java Sample Server, and also presents an example of its use. Finally, the seventh chapter presents conclusions.

# 2 OPC Unified Architecture

This section is an overview of the OPC Unified Architecture specification. First, a concise overview will be presented. Next, the development of OPC UA and its relation to classic OPC will be explored. Also its technological base and potential capabilites are explained, and finally the current status and future of OPC UA is detailed.

## 2.1 Overview

OPC Unified Architecture, or OPC UA for short, seeks to build on the strengths of classic OPC while fixing its flaws and adding new features. OPC has been very successful since its launch in 1996, reaching the status of a de facto standard in a matter of years.[2] However, it does not meet all the requirements of today and thus development of replacement was begun. Development of OPC UA was begun after the limitations of classic OPC had become apparent, and the first parts were published in June 2006.

The two main components of OPC UA are transport mechanisms and data modeling.[2] Because communication is no longer tied to any propietary technology (as OPC was to COM/DCOM and thus Microsoft), an important improvement in UA is that it is not operating system dependent or language dependent. While OPC Foundation provides reference stack implementations written in ANSI C, C# and Java, this is by no means a restriction. OPC UA stacks can be written in any language and run on any operating system/environment. This means that it is possible to implement UA applications in, for example, embedded devices. Also, with the rising popularity of Linux-based systems, not relying on the Microsoft COM/DCOM communication protocol is a key point. UA uses non-propietary technologies for communication and is not communication protocol dependent. The two protocols implemented in the OPC Foundation SDK[3] are TCP and Web Services, but additional protocols can be defined in the future. Data modeling is the second base component of OPC UA, and is the focus of this thesis.

An important difference between classic OPC and OPC UA is the range of applications for which they can be used. OPC was originally developed to transfer data in process industry applications, and while it has also been successfully used in other automation applications, such as discrete manufacturing, it doesn't fit other data transfer applications very well. This has been rectified in OPC UA. Different application fields, including those that were not supported in OPC such as data exchange between ERP systems, can be accommodated.[2] The semantics of the data are not pre-defined like they were in classic OPC, which means that information can be modelled and context provided for data.

## 2.2 History and relation to OPC

OPC UA is the successor to the the original OPC specification. OPC is a major communication standard for automation applications, first published in 1996 and maintained by the OPC Foundation. OPC eliminated the need for separate drivers for each vendor[4], and for the first time enabled interoperability in automation systems. Because OPC was restricted to COM-based communication and because it focused on important features, it allowed quick adoption and thus quickly became a de facto standard, which it remains up to date.[2]

OPC has a few serious limitations. It is based on Microsoft's propietary COM/DCOM component technology, which not only restricts the platform on which OPC can be implemented to just Microsoft Windows, but COM/DCOM is being phased out in new Microsoft products as well. Classic OPC doesn't handle network connections all that well and it also lacks security features. Because OPC was defined in parts and collaboration between the groups and technologies was not tight, the different OPC facets (such as Data Access, Alarms & Events, etc.) use expose their services in different ways. With regard to information models, OPC doesn't support complex data models or address spaces, as it was designed for representation of basic automation information.[5]

To rectify these problems, the development of OPC UA was begun in the early 2000's. The first parts of the Unified Architecture specification were published in 2006. Because of its open-ended and configurable nature, it is to be expected that OPC UA will also spread to applications to which classic OPC was not suited.

## 2.3 Technology and capabilities

This section will explain the basic concepts and technology of OPC UA.

### 2.3.1 General

The name of OPC Unified Architecture refers to it exposing all the facets of OPC using a single set of services, so that one data model serves all the use cases. UA follows the SOA (Service Oriented Architechture) paradigm, so all functionality is based on the concept of services.[5] Services are defined in the OPC UA specification as "Client-callable operations in an OPC UA Server".[6] A server may implement all services defined by OPC UA or just a subset of them.

OPC UA is language and operating system independent, which enables a multitude of implementations on different platforms. The basic implementation of OPC UA which implements standard features needed for successful communication is called a stack. A stack contrasts with a SDK in that using it requires substantial knowledge of the OPC UA specification and an SDK hides as much of the complexity underneath it as possible. OPC Foundation provides a .NET stack written in C# to its members (which also includes a version in ANSI C), and also distributes a Java stack

combination. Also at the time of this writing it is known that a full-fledged implementation in C++ exists, made by Unified Automation GmbH of Germany. A stack can be, in principle, written in any language that supports the basic features necessary for implementation. Although the OPC UA Address Space model is heavily object-oriented, object-orientedness is by no means required of the implementation language, like the implementation in ANSI C proves. The language-independence also means that implementations are not restricted to Microsoft Windows platforms, like classic OPC was due to the use of COM.[2] Hannelius et al. also document porting the ANSI C stack to ARM9 architecture and creating an OPC UA server in an ARM9 based single board computer.[14] This demonstrates that UA can be used in widely varying environments.

The most basic concepts in OPC UA are transport mechanisms and data modeling. For transporting data, OPC UA can use any protocol, but the first version has two technology mappings, a binary one called UA TCP and a Web Service based protocol. Transport mechanisms are not in the scope of this thesis and will not be discussed further.

Data modeling defines the basic tools for exposing information in OPC UA. The specification defines a base Address Space which contains entry points and base types which are used to build a type hierarchy, which may also be extended by vendors or other organizations. Information models build upon the base abstract modeling concepts.[2]

### 2.3.2 Address Space

Information in OPC UA is expressed as an AddressSpace, defined as "the collection of information that an OPC UA Server makes visible to its clients".[6] The fundamental component of an AddressSpace is a Node, which are used to represent all information within the server. Nodes are connected to each other by a variety of References. Examples of References are the Organizes reference type which is used to build hierarchies in the AddressSpace, and the HasTypeDefinition reference which is used to define the instance Nodes' types. The AddressSpace is an interconnected mesh of Nodes connected by various references.

The basic structure of the AddressSpace is based on Objects, which typically represent real-world entities and serve as a way to organize information. The basic facets of an Object are represented in Figure 1. An Object can contain Variables, Methods which are used to invoke actions on the Object and Events, which notify a listener of alarms, events etc.

Each OPC UA server AddressSpace has some common nodes. A basic Objects hierarchy, depicted in figure 2, forms the basis for the server's Object hierarchy.

These standard nodes can be used as entry points to the AddressSpace when browsing it, as they are guaranteed to exist on all OPC UA servers. They can be thought of as folders which organize the AddressSpace according to the NodeClass of the Node. There are eight NodeClasses which are used in the AddressSpace, and a cer-

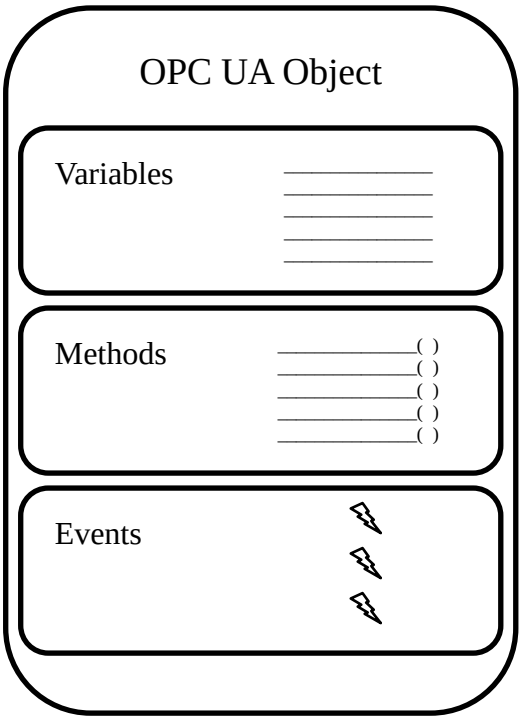


Figure 1: OPC UA object.[15]

tain Node can only belong to one of them. Each of the eight NodeClasses has its own distinct purpose, which are explained below.

**Objects** represent real-world entities, such as specific valves. They also are used to organize the AddressSpace, as with the base hierarchy which is composed of Objects.

**ObjectTypes** are abstractions of Object entities, like a certain model of a valve. The ObjectType will define the structure all the Object instances will have. Structure in this case means all child Nodes of the Object.

**ReferenceTypes** defined the semantics of References between Nodes. Each Reference has a target and a source Node in the AddressSpace (or optionally, in some other server's AddressSpace), and the ReferenceType defines what the relation between the Nodes is. ReferenceTypes can also be subtyped, and there are abstract ReferenceTypes which cannot be instantiated, only the subtypes of them.

**Variables** hold data related to objects, and may have sub-variables of their own. Properties are a special type of Variable, and differ from Variables in only that they cannot have any child Nodes.

**VariableTypes** relate to Variables as ObjectTypes to Objects.

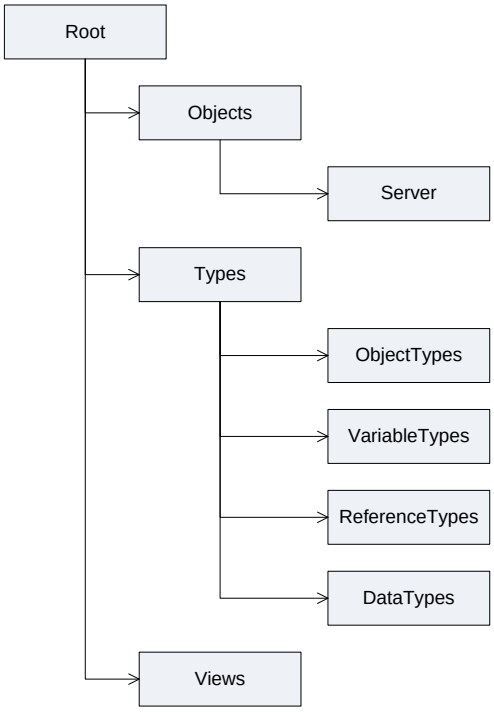


Figure 2: Basic Object hierarchy of OPC UA servers.[15]

**Methods** are callable operations, and they are children of an Object and represent possible operations with the Object in question. For example an Object representing a pump might have a Method to start it and another one to stop it.

**DataTypes** are the basic types the data is represented in, such as integers, strings, etc. New DataTypes may also be defined.

**Views** are subsets of the whole server address space which are used to reduce clutter if only a specific part of the AddressSpace is needed for some purpose. For example, one View could be used for maintenance and another for configuration, so those parts of the Address Space that aren't needed in the task at hand are not visible. View nodes represent them in the AddressSpace.

All nodes in the AddressSpace have a NodeClass, and NodeClass is not extensible, that is, no new NodeClasses may be defined. The different NodeClasses and their Attributes are shown in figure 3.

The BaseNode presented in the figure is not a real NodeClass, but a representation of the common Attributes which all NodeClasses share. These Attributes are presented in table 1. The NodeClass field is mandatory, as are a NodeId, a BrowseName and a DisplayName. The difference between BrowseNames and DisplayNames is that since BrowseNames are used for browsing the AddressSpace, all instances of a type

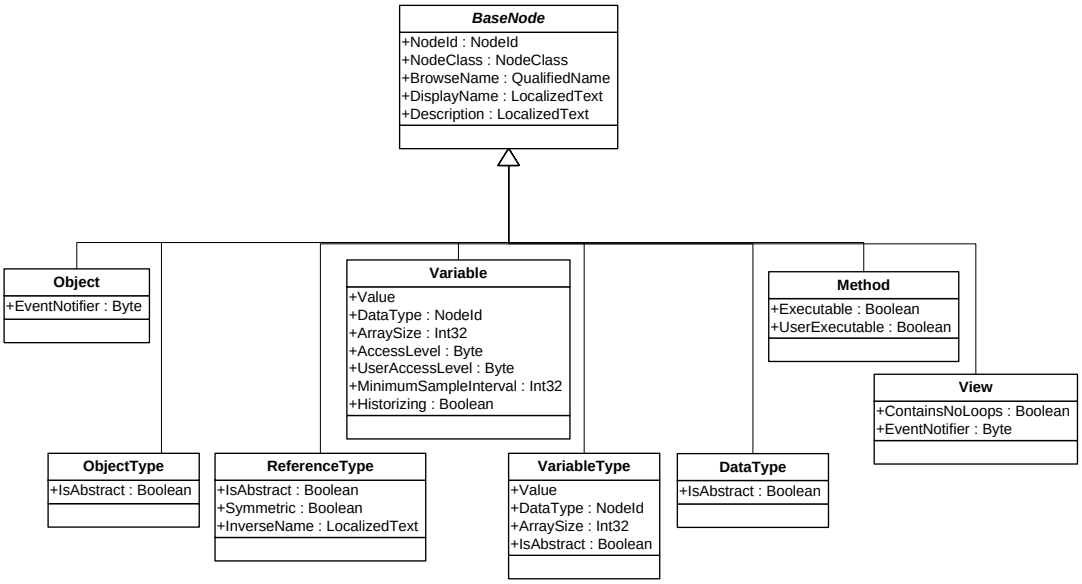


Figure 3: The eight NodeClasses and their abstract parent.[16]

will have the same BrowseNames, whereas DisplayNames are not restricted and they are what is supposed to be shown to an operator. In addition, a Description for the Node may be specified, as well as a WriteMask and an UserWriteMask which have to do with writing rights to the Node. In addition, some NodeClasses have additional Attributes, for example Variables have an Attribute named Value, which holds the value of the Node. Attributes are elementary components of NodeClasses, and they may not be extended.[7]

Table 1: BaseAttributes common to all nodes.

| **Attribute** | **Type** | **Use** |
|---|---|---|
| NodeId | NodeId | Mandatory |
| NodeClass | NodeClass | Mandatory |
| BrowseName | QualifiedName | Mandatory |
| DisplayName | LocalizedText | Mandatory |
| Description | LocalizedText | Optional |
| WriteMask | UInt32 | Optional |
| UserWriteMask | UInt32 | Optional |

Perhaps the most important attribute Nodes have is the NodeId. Each node is identified by an unique NodeId, which consists of three parts: an address space index, which can be translated into an address space URI by the server, an enumerated identifier type, and the identifier itself. The identifier can be one of four types:

numeric, string, GUID (Globally Unique Identifier) or opaque (in practice, a byte table). Another type of node identifier is the ExpandedNodeId, which has an address space URI in addition to the index. The URI is optional, but if present, the index will be disregarded.[10] Nodes are identified in all Service requests by their NodeId.

### 2.3.3 Services

OPC UA Part 4 is about Services, which "are the collection of abstract Remote Procedure Calls (RPC) that are implemented by OPC UA Servers and called by OPC UA Clients."[8] In a sense, Services are the core of OPC UA, since all interactions between UA clients and servers happens through the use of these services.[8] Services in OPC UA are designed for bulk operations to avoid roundtrips.[2] Each service has parameters that are sent to the server in the service call, and the server responds to the call with results, typocally telling if the service call succeeded or not. If the service call failed for some reason, this reason is specified.

Services are further categorized into sets of similar services, called Service Sets. Each service set is used by the client to accomplish specific tasks, like the Attribute Service Set for reading or writing attributes, including values, or the Method Service set for using methods on the server. Table 2 lists the ServiceSets and their Services.

Table 2: OPC UA Services.

| **ServiceSet** | **Services** |
|---|---|
| Discovery | FindServers, GetEndpoints, RegisterServer |
| SecureChannel | OpenSecureChannel, CloseSecureChannel |
| Session | CreateSession, ActivateSession, CloseSession, Cancel |
| NodeManagement | AddNodes, AddReferences, DeleteNodes, DeleteReferences |
| View | Browse, BrowseNext, TranslateBrowsePathsToNodeIds, RegisterNodes, UnregisterNodes |
| Query | QueryFirst, QueryNext |
| Attribute | Read, HistoryRead, Write, HistoryUpdate |
| Method | Call |
| MonitoredItem | CreateMonitoredItems, ModifyMonitoredItems, SetMonitoringMode, SetTriggering, DeleteMonitoredItems |
| Subscription | CreateSubscription, ModifySubscription, SetPublishingMode, Publish, Republish, TransferSubscription, DeleteSubscriptions |

Regarding the focus of this thesis, the most interesting Service Set is NodeManage-

ment, which allows a client to add and delete nodes and references in the server's AddressSpace. The NodeManagement Service Set consists of four Services, AddNodes, AddReferences, DeleteNodes and DeleteReferences.[8] These services will mainly be used by the configuration clients of UA servers.[2]

**AddNodes** is used to add one or more Nodes into the server's AddressSpace hierarchy. This service ensures that all added Nodes are the target of a HierarchicalReference, thus ensuring that the AddressSpace stays connected.[8]

**AddReferences** is used to add References to one or more Nodes in the server's AddressSpace.[8]

**DeleteNodes** is used to delete one or more Nodes from the server's AddressSpace. The service call may specifies if those References with deleted Nodes as targets should be deleted as well or not.[8]

**DeleteReferences** is used to delete one or more References from the server's AddressSpace.[8]

The Services used to delete have some special considerations. In situations where Nodes from an instance of a type are deleted (effectively making the type definition invalid) or Node with children is deleted without its children being included in the service call, it is unclear whether more Nodes than the one specified should be deleted. In principle, any client calling these operations must carry the responsibility for these operations and the server must not block the operations. For example, if a Node in the AddressSpace, which has child Nodes, is deleted in order to replace it with another, it must be possible to delete it even while it makes the AddressSpace momentarily unconnected. The same applies to deleting References.

In principle, all modifications to the AddressSpace can be carried out using these four Services. The NodeManagementServiceSet does not offer tools to control how the Nodes may be bound to an underlying data system, so it is something that needs to be taken separately, as discussed later in this thesis.

## 2.4 Current status and future

At the time of writing this, OPC Foundation has released ten parts of the OPC UA specification. In addition one part, Alarms and Conditions, has been released as a release candidate. In addition, two information model definitions called companion specifications have been released and one is also available as a release candidate. The companion specification define an OPC UA information model for a specific application field, and are detailed further in section 4.4. Partitioning is required for IEC standardization, as OPC UA will be known as IEC 62541[2]. The different parts of the specification, along with their status, are presented in table 3. Parts 1-7 define the core of OPC UA while parts 8-11 define access types[2]. Parts 12

Table 3: The current parts of the OPC UA specification.

| Part | Name | Version |
|---|---|---|
| 1 | Overview and Concepts | 1.01 |
| 2 | Security Model | 1.01 |
| 3 | Address Space Model | 1.01 |
| 4 | Services | 1.01 |
| 5 | Information Model | 1.01 |
| 6 | Mappings | 1.00 |
| 7 | Profiles | 1.00 |
| 8 | Data Access | 1.01 |
| 9 | Alarms and Conditions | 1.00.18 (RC) |
| 10 | Programs | 1.00 |
| 11 | Historical Access | 1.00 |
| 12 | Discovery | Not released |
| 13 | Aggregates | Not released |
| | | |
| | Devices | 1.00 |
| | Analyser Devices | 1.00 |
| | OPC UA for IEC 61131-3 | 0.09.1 (RC) |

and 13 have not been yet released. The three companion specifications are shown separately.

With the recent release of several SDKs and software products, it is to be expected that OPC UA will gradually start taking foothold in new industry installations. The transition to UA is likely to be slow, because in many current installations there is no reason to upgrade to newer technologies. However, this problem is somewhat alleviated by wrappers which allow a classic OPC server to function as a OPC UA server, but there are drawbacks related to maintainability and security related to this approach.[17] Migration to OPC UA can be done either with using wrappers to allow OPC UA clients to contact OPC servers, or by implementing native OPC UA servers. These two migration strategies are presented in Figure 4.

OPC UA widens the application field of classic OPC. UA can be used in communications between almost any conceivable information systems, and thus it is expected that UA will take a foothold also in the upper tiers of the corporate information systems hierarchy, unlike OPC which was only in the lower layers. UA is applicable for manufacturing software applications in field devices, control systems, MESs and ERPs[6], as illustrated in figure 5. However, there is some disagreement over the suitability of OPC UA to the very highest tiers of the corporate information pyramid, meaning for example communication between ERP systems. Dennis Brandl expressed his view at OPC Day 2009[18] that OPC UA may not be currently suitable for cross category communications between manufacturing operations systems, such

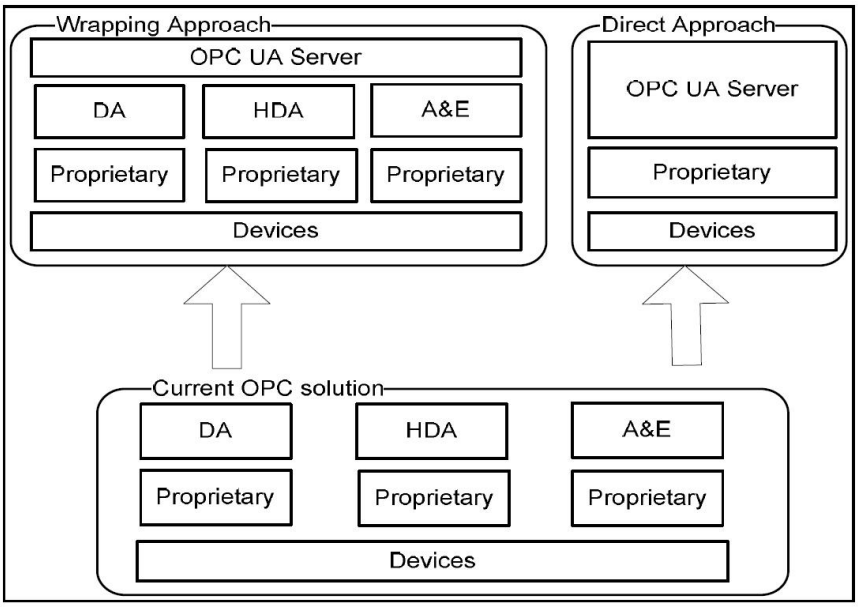


Figure 4: Migration strategies.[17]

as MESs (Manufacturing Execution Systems). Even more importantly, his view is that OPC UA may be ill-suited to communications with ERP systems, since OPC UA communication is inherently tightly coupled, whereas ERP communications are loosely coupled and message based. Nevertheless, Brandl agreed with the consensus on OPC UA being well suited to communications lower in the pyramid.

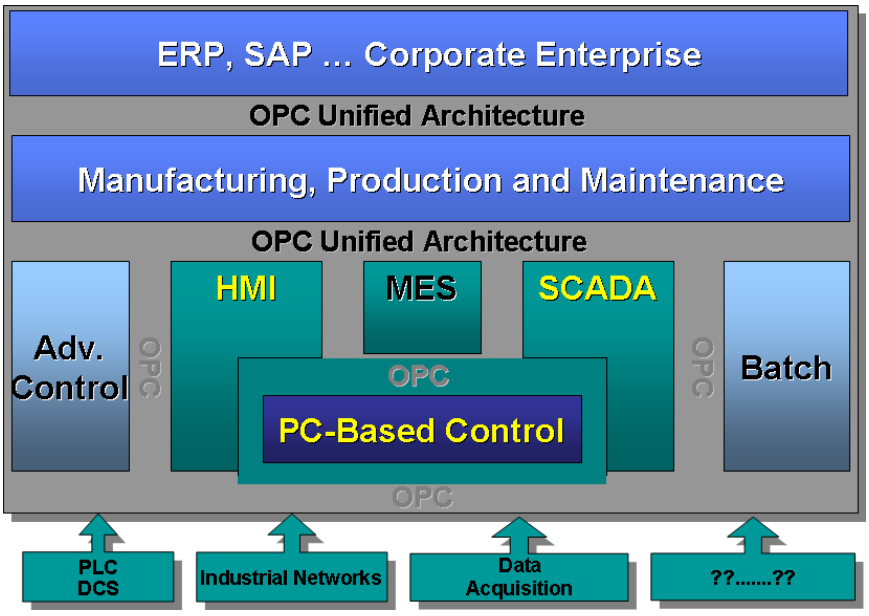


Figure 5: OPC UA applications.[6]

# 3 Implementations of OPC UA

## 3.1 Introduction

OPC UA in itself is a set of specifications, meant to provide different implementations of servers and clients with common ways to operate. The actual implementation is of course up to the vendor in question, but a common ground is provided by shared communication stacks and software development kits (SDKs). Typical software layers are presented in figure 6.

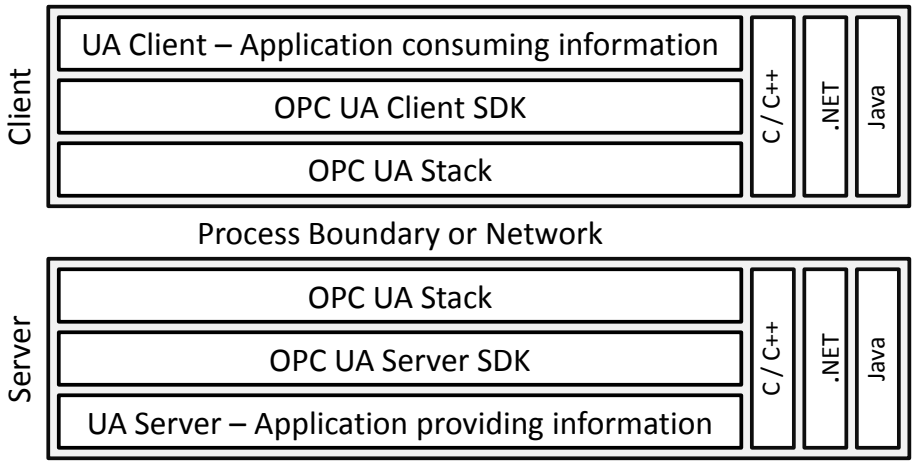


Figure 6: OPC UA software layers. (Adapted from [2])

OPC UA applications (clients and servers) use the interface provided by the SDK to communicate to each other, but actual UA communication takes place between the stacks. Both stacks and SDKs can be implemented in a variety of languages and platforms, of which three of the most relevant languages, C/C++, Java and .NET are shown in figure 6. With these three implementations maintained by the OPC Foundation, the main environments and programming languages are covered[2].

A communication stack is a collection of code that enables an application to communicate via OPC UA. It implements the basic features of OPC UA, called Services. Stack implementations can be difficult to use or understand and are likely to require thorough knowledge of the specifications themselves in order to work properly. The basic functions a stack implements are message encoding, message security and message transport[2]. This means that many of the practical necessities, such as handling subscriptions and sessions, are left for the application developer to implement. Developing application right on top of the stack requires a lot of of extra work, which is why SDKs are used. The only downside that using SDKs has is that for very low performance embedded servers it might be necessary to develop custom optimized solutions as the basic SDKs may be too demanding for their resources.

SDKs are a software layer built on top of the stacks, meant to handle low-level tasks

and hide unnecessary complexity from the application programmer, exposing a less detailed and easier to use API (Application Programming Interface, the interface that is exposed to the application programmer). They are built on top of a stack, but hide much of the underlying mechanisms of OPC UA that are common to all UA applications. As is summarized in the documentation of Unified Automation's C++ SDK, "a SDK simplifies the UA stack APIs, implements common UA functionality needed in most or all UA applications, provides base functionality and helper functions, implements the security handling and provides samples for common use cases."[20] They are meant to enable a developer to create an application for OPC UA without requiring knowledge of the specifics of transfer protocols, encoding, session establishment etc. SDKs also offer samples to speed up and ease their use. Samples are important because making new applications using only the documentation as reference can be harder than seeing working examples and building on them. SDKs are thus a library for creating applications, and in UA issues they handle are things like security certificates, session handling and taking care of subscriptions. More advanced SDKs also enable RAD (Rapid Application Development) techniques, with a great deal of easily reusable code. This is done by implemeting several necessary functionalities straight out, while allowing the behavior to be modified in order to accommodate needs for custom functionality.

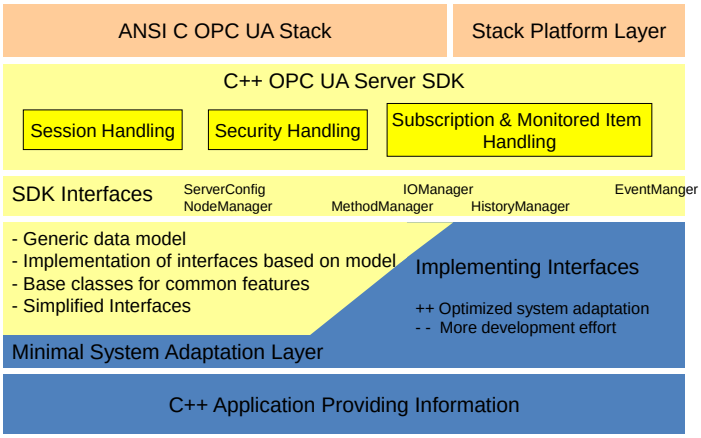


Figure 7: SDK acts as a middle layer between stack and application[21]

Figure 7 presents an overview of the SDK between the stack and the application. On the top is the stack, in this case the ANSI C stack. On the bottom is the application itself, and between these two is the SDK. The top part of the SDK, which is in all yellow, is the constant part that is always in use. It handles tasks common to all UA applications, such as session, security and subscription management. Below it is a bar that represents an internal interface in the SDK. The interface is there to allow custom implementations of certain components to better accommodate the underlying application. On the left side, no custom implementations have been made and only a thin layer acts as an adapter between the SDK and the application.

This means using standard implementations of the customizable components. On the right side, custom implementations have been provided for the customizable components.

The following sections will present three stacks and four SDKs to whose source code the author has had access. OPC Foundation distributes stacks for .NET (implemented in C#), ANSI C and Java. It also distributes a .NET SDK to its members. A Propietary SDKs have been developed for Java by Prosys PMS Ltd[19] and for ANSI C and C++ by Unified Automation GmbH[20]. The following section will present all of them and make some comparisons.

## 3.2 Stacks and software development kits

This section will present three stacks and four SDKs on a general level. A few things that are relevant with regard to this thesis are explained more closely. First, the way the address space is implemented in each of the stacks/SDKs, and second, how information models are accommodated by the stack/SDK.

### 3.2.1 OPC Foundation .NET SDK

OPC Foundation distributes a .NET stack and SDK[3] to its members. The stack features support for both binary and Web Service transport protocols and it can be regarded as the reference implementation for OPC UA. The distribution also includes the ANSI C stack, and applications have a choice of which stack to use. The SDK also includes straightforward tools for interoperation with OPC servers and clients, as it includes wrapper and proxy components for using OPC.

In this implementation, the Address Space is handled by NodeManagers which are governed by a MasterNodeManager. The main classes that take part in node management in the .NET SDK are presented in figure 8. The server object is shown in the figure as StandardServer, which is a base implementation of the server object that can be inherited from in custom server implementations.

It owns two special NodeManagers, the MasterNodeManager and the CoreNodeManager. The MasterNodeManager also uses the CoreNodeManager, and in addition, it owns all the other NodeManagers in the server application. The MasterNodeManager acts as a sort of manager for the NodeManagers, that is, service requests that the server receives are passed on to the MasterNodeManager, which in turn relays them forward to the NodeManagers. Each NodeManager owns a certain set of nodes. Combined, the sets of Nodes make up the whole AddressSpace of the server. The CoreNodeManager is a special standard NodeManager, which can be used to govern all Nodes that have no special needs. A good example of a situation which would warrant creation of a custom NodeManager is when a set of Nodes represents data from some underlying system. Then the task of the NodeManager would be to act as the middleware between OPC UA and the underlying system. This is how the OPC wrapper has been done in the SDK, it is actually a NodeManager which

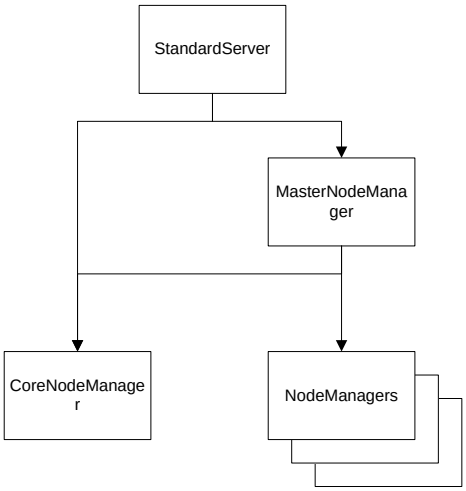


Figure 8: .NET server object and its NodeManagers

connects to an OPC server and makes its data available in the OPC UA server.

The .NET SDK supports deploying information models in the server by using a code generator, UA Model Compiler, which is bundled with the SDK. XML files of a well-defined format are given to the tool as input, and it generates code from them. This code can then be deployed with the SDK to create a server which includes the information model. The XML format used with the UA Model Compiler is also used for defining information models in this thesis, and is discussed further in section 6.3.3. Using C# partial classes, it is possible to customize the behavior of the generated classes while also being able to make modifications to the source XML and regenerate the classes at any time.

The .NET SDK also supports custom code in the server with event handlers. Nodes can have an event handler tied to service calls for the most common cases, such as reading and writing values and history data. This allows comparatively easy overriding of these functions, while allowing the server to do the rest based on standard functionality. Custom code may for example be used to check validity of values to be written. Using event handlers is also an alternative way of binding underlying system data to the Nodes.

### 3.2.2 Unified Automation C++ SDK

Unified Automation has developed a C++ SDK on top of the ANSI C stack developed by OPC Foundation.

The NodeManagers differ from the .NET implementation somewhat. In essence, the duties that a NodeManager performs in the .NET SDK are done by two classes in the C++ SDK, NodeManagers and IOManagers. NodeManagers provide browsing and managing the AddressSpace, while IOManagers provide reading, writing and

monitoring data.

This SDK also includes a tool for creating information models and generating code for them. It is intended to support multiple languages and be user customizable. A more detailed account it is found later in section 5.5.2 of this thesis when exploring different information model implementations.

### 3.2.3 Java Stack and Toolkit

A consortium of Finnish enterprises, one of which is Prosys PMS Ltd, has collaborated on a Java stack for OPC UA. This stack has some higher-level features, such as implementations for some higher level concepts such as nodes (which for example the ANSI C stack does not) but lacks a straightforward API for developing applications. The Java stack is now governed by OPC Foundation.

In addition, a Java Toolkit has been created as a prototype SDK for developing UA applications in Java. It is an internal result of the project group and not meant for general distribution. It features many of the same features as well as the general structure of the .NET SDK. In particular, the way the NodeManagers are implemented is very similar to the one in the .NET SDK. The Java Sample Server, for which the information model implementation in this thesis has been developed, is a part of the Java Toolkit.

### 3.2.4 Prosys OPCUA Java SDK

Prosys has developed a server/client SDK on top of the Java stack distributed by OPC Foundation. The Java SDK wraps the stack and simplifies the API compared to the stack, so that adding OPC UA functionality to new or existing Java applications is simplified compared to using the base stack. At the time of this writing, the client SDK has just been released and the server SDK is at beta. The server SDK will incorporate lessons learned during the writing of this thesis.

## 3.3 Availability and licenses

The .NET, ANSI C and Java stacks are distributed by the OPC Foundation to its corporate members. The files are under four licenses, with binaries and documentation files distributed under OPC Redistributables License and source files under OPC Foundation MIT License, OPC Reciprocal Community License or OPC Reciprocal Community Binary License depending on the files.

The propietary SDKs are commercially available from the companies that developed them and have their own licences.

# 4 Information modelling

This section defines an information model in the context of this thesis, and presents the base OPC UA information model. In addition, extensions to the base information model will be presented, and finally a few thoughts about the semantic nature of the models will be discussed.

## 4.1 Motivation for using information models

Although OPC UA offers a lot of tools and great potential for using information models, their use is not necessary. In simple applications, it is possible to construct an address space in the server and use it efficiently with only very rudimentary typing and modelling of the exposed information. This way, the data transport capabilities of UA are harnessed and data remains very close to what it was in classic OPC, that is, just pure data. However, by using information models, clients can perform sophisticated tasks by interpreting the semantics of the provided information, and not just the values.[2]

The main advantages and principles of information modelling in OPC UA are as follows according to Damm et al[2].

- Object-oriented techniques, such as type hierarchies and inheritance, allow clients to handle all instances of a certain type in the same way while allowing them to ignore unnecessarily specific information.
- Type information is exposed and accessible in the same way as instances.
- The network of nodes, being full-meshed, allows for several hierarchies in the address space.
- Extensibility of types and reference types creates an extensible information modelling environment.
- Any information model can be exposed in UA, so that systems that have an information model defined do not need to be mapped to another model.
- All information modelling is on the server-side, so it is not necessary for clients to support it.

All in all, the support for information models will enable new kinds of applications to be developed.

It is to be expected that for most applications, application domain specific infomation models such as the abovementioned Devices companion specification will be available when developing a new information model. Thus the development of the new information model will be in fact extending the existing application domain model. An example of an information model built upon the Devices specification is

presented in Matti Nykänen's thesis[22]. He defines a generic information model for electric drives, using the Devices information model as a base. Using this information model framework specific information models for different drive types can be defined, while maintaining interoperability on several levels.

With most new solutions, developing an information model to represent the application domain is necessary. If it is not possible to use a domain specific information model such as the UA Devices model (because such a model does not exist or it is ill-suited to the purpose at hand) as the basis for the new information model, one must be defined from the ground up (or in other words, using the UA Address Space Model as base). The disadvantage of this is that the advantages of interoperability which using common base models confers are lost.

## 4.2 Information models

An information model, in the context of OPC UA, is defined as "an organizational framework that defines, characterizes and relates information resources of a given system or set of systems". [6] Information models can be layered on top of each other, as seen in Figure 9. The basic layer is on the bottom, and represents the basic services all information models build on. On top of these there are the Data Access, Alarms & Events, Historical Data Access and Commands models, which are parts of the UA specification. Different organizations can define application domain specific models on top of these two layers. Finally, vendors can define their own information models, e.g. for specific devices, on top of the previous layers.

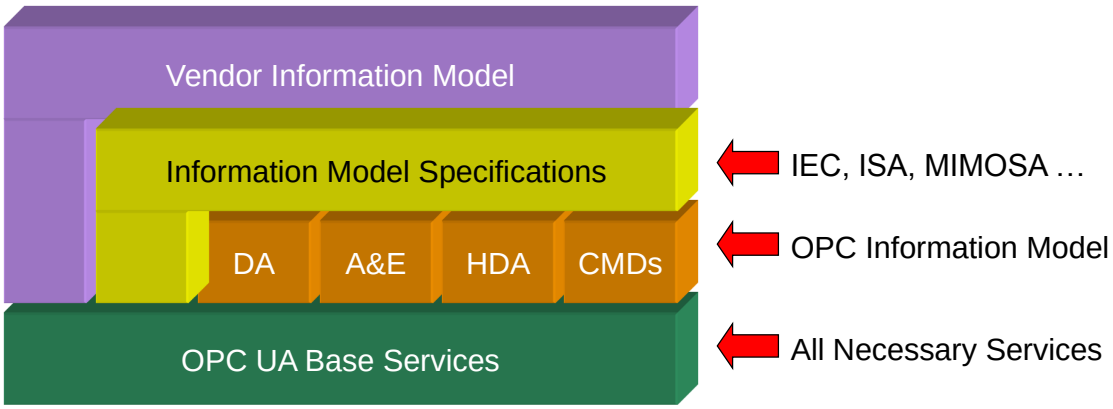


Figure 9: OPC UA information model hierarchy.[15]

The lowest layer in Figure 9 is the OPC UA Address Space model, defined in OPC UA Specification Part 3: Address Space Model[7]. It is the meta model on which all other information models are based on and also the information model of an empty server, which means that all OPC UA servers have it. The next layer contains some

convenience models for specific operations, such as Data Access and Commands. These inherit their form from classic OPC, but build on the basic Address Space model to provide their services. The layer on top, shown in yellow, represents different application domain specific models released by well-known authorities, such as standardisation organisations. An example of this layer are the previously mentioned companion specifications. Their purpose is to define common information models for applications in specific industry fields, so that interoperability within the industries is maximised. Application domain specific information models can build on both the basic Address Space model and the models built upon it. Finally, on top of all of these, vendors can define their own information models to supplement the lower levels. When considering the layer model, it is important to note that clients do not necessarily have to be previously aware of the higher information models but can still discover all data from them, and that not all servers will have all the layers in their Address Space. For example, it is possible to create an embedded server whose Address Space consists solely of the base Address Space and a vendor-specified information model. Multi-use servers will have larger and more complex address spaces consisting of several information models. Different Information Models in one server are supported by the use of NamespaceIndexes[16]

An information model based on OPC UA is defined in the following steps according to Mahnke[16].

- Define Types, such as Object Types, Variable Types, Reference Types and Data Types
- Define standard Methods
- Define Properties
- Define Modelling Rules
- Define Encoding for Data Types to transfer data directly in the format that is needed

According to Mahnke, this mechanism is already used in parts 5 and 8-11 of the OPC UA specification to define information models. Types serve as the basic building blocks of the AddressSpace, and they will have Methods, Properties and Modelling Rules attached to them. Modelling Rules are a mechanism for defining how Types can be instantiated and inherited. Subtypes may only tighten constraints of the parent type. DataTypes are used to define simple and complex types for data values. DataTypes will also have an encoding defined, which allows them to be efficiently transported. For built-in DataTypes and their subtypes (which are encoded like their supertypes and thus considered simple types), this encoding will not be presented in the AddressSpace, since it can be assumed that all OPC UA applications are familiar with their encoding. For custom complex DataTypes, encodings are defined separately and are presented in the AddressSpace.[7]

## 4.3 Representations of information models

As all information is modelled in OPC UA as Nodes which reside in the server's Address Space, the information models are also exposed to the clients in the form of type definitions. While there may be several instances of a certain type in the system, they will each link to the same type definition. The HasTypeDefinition ReferenceType is used for these references. Clients can be programmed against type definitions, so that each instance can be treated the same way. The formal notation for OPC UA Information models is exposed in Figure 10.

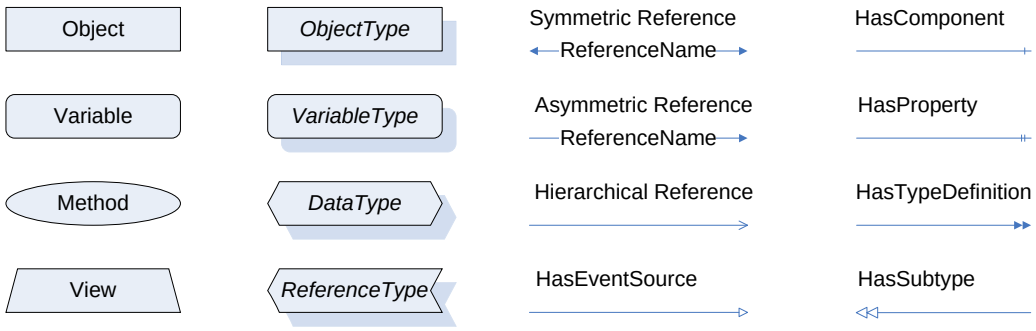


Figure 10: OPC UA information model notation.[16]

Each of the NodeClasses detailed in section 2.3.2 have their own symbol in the notation, and there are also eight different kinds of arrows to represent different kinds of references. The following is an explanation of the ReferenceTypes in figure 10, based on Part 3 of the OPC UA Specification[7]. It it to be noted these are but a small cross-section of the most important predefined ReferenceTypes.

**Symmetric References** are those References whose 'Symmetric' Attribute is set to true. This means that the meaning of the Reference is the same when viewed from the source Node or the target Node.

**Asymmetric References** are those References whose 'Symmetric' Attribute is set to false. This means that the meaning of the Reference, when viewed from the target Node, is the inverse of the meaning when viewed from the source Node. The inverse meaning may be named using the Attribute 'InverseName'

**Hierarchical References** are used to define hierarchies in the AddressSpace, though it does not preclude loops. It is an abstract ReferenceType.

**HasEventSource** is used to build non-looping hierarchies that relay events. If a client listens to events from a Node in the hierarchy, it shall also receive events from all Nodes beneath the listened Node in the hierarchy. It is a concrete ReferenceType, which means it may be used directly in the AddressSpace.

**HasComponent** is used for references which define the target Node of the reference to be a part of the source Node. It is a concrete ReferenceType.

**HasProperty** identifies Properties of a Node. It is a concrete ReferenceType.

**HasTypeDefinition** binds Objects or Variables to their respective ObjectTypes or VariableTypes. It is a concrete ReferenceType.

**HasSubType** is used to express subtype relationships in the type hierarchy. For example, abstract ReferenceTypes can have concrete subtypes using this ReferenceType. It is a concrete ReferenceType.

The notation shown in Figure 10 is the one used by the OPC Foundation, but of course other graphic representations are also valid. The OPC UA notation is actually stereotyped UML[2] and is detailed in UML format in Annex B of Part 3 of the OPC UA Specification[7]. An example of a type presented in the notation discussed above is presented in Figure 11. It is an ObjectType that represents a boiler. The boiler has three components, PipeX001, DrumX001 and PipeX002, which are referred to from the boiler by using the HasComponent ReferenceTypes. Water flows from PipeX001 to DrumX001 to PipeX002, and this has been represented using a custom ReferenceType, FlowTo. Each of the components has subcomponents, like FTX001, and these in turn have Variables. In addition, on the right side, the boiler's control structure is displayed. There are three objects, FCX001, LCX002 and CCX001, which have named subvariables. The control structure is also defined using HasComponent references, which is a hierarchical ReferenceType. There are two hierarchies in the type, one for the physical components of the boiler and another for the control structure. These two hierarchies are linked together using a custom non-hierarchical ReferenceType, Signal. In the figure, it is easy to understand the connection; FTX001's DataItem is used as Measurement for the control circuit FCX001, whose ControlOut is used as the DataItem of ValveX001, in other words, its position.

Although typically an information model may be designed as a picture to be easily understandable by model designers, use in applications requires it to be represented in machine-readable form. For this XML (eXtensible Markup Language) documents are a fitting solution, and the OPC Foundation .NET SDK uses an XML file to define the basic UA information model. XML is a widely used markup technology and as such, many programs and program libraries support its use. Due to the benefits of using XML to define models, they will be used with all likelihood in future applications and also in this thesis. The .NET SDK XML format is further explained in section 6.3.3.

## 4.4 Companion specifications

In addition to the base information model, the OPC UA Address Space Model, that is a part of the OPC UA specification and the vendor-specific information models,

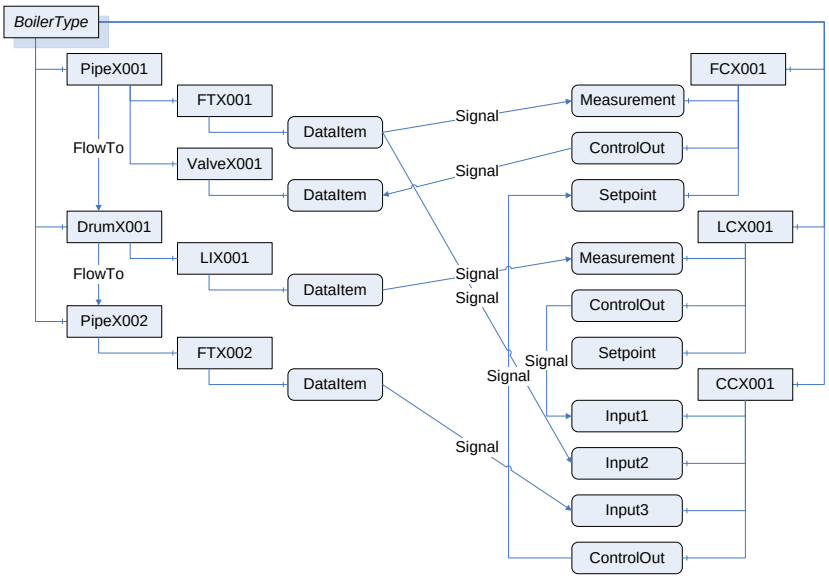


Figure 11: An example of an ObjectType definition.[16]

there are a number of organization-defined information models called companion specifications. These model specific application domains and thus extend the UA standard, and are intended to "permit industry groups to define how their specific information models are to be represented in OPC UA Server AddressSpaces"[6]. They support the spirit of interoperability by unifying the representation of information in an application field. This section lists some of these information models.

The Devices information model[11] and its extension, Analyzer Devices information model[12] will be briefly presented next. They are so-called companion specifications to the OPC UA specifications, and are good examples of application domain specific information model frameworks. It is to be expected that they will serve as the basis for several other information models. Leitner and Mahnke[17] mention that standards like EDDL (Electronic Device Description Language) and FDT (Field Device Tool) will eventually take advantage of the opportunities presented by OPC UA and will define their own companion specifications to OPC UA, modeling their own domain-specific information models to be accessible via OPC UA. Since there is little concrete information available regarding these models yet, they will not be detailed in this thesis.

### 4.4.1 Devices information model

The Devices information model is a part of the OPC UA specification that is intended to unify the way automation devices are represented in OPC UA. At the time of writing the specification has just been released as version 1.00. The information model described is meant to provide a unified view of devices, regardless of the device protocols actually used. In the scope of the information model, a device is defined as "an entity that provides sensing, actuating, communication, and/or control functionality". The information models defined using the Devices specification are mostly composed of Object and Variable NodeClasses.[11]

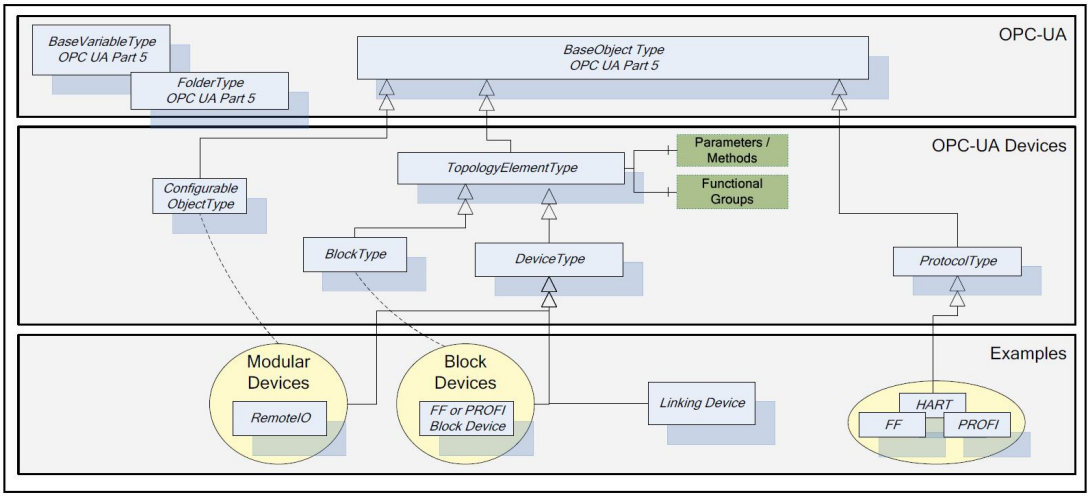


Figure 12: Overview of Devices information model.[11]

Figure 12 presents an overview of the types in the Devices information model. The specification adds some types, such as TopologyElementType, ConfigurableObjectType and ProtocolType. The base ObjectType for device topology elements is TopologyElementType, from which BlockType and DeviceType inherit. TopologyElements may have Parameters and Methods, which (if the object is question has them) are kept in flat lists called ParameterSet and MethodSet. FunctionalGroups are used to organize these elements according to the structure of the TopologyElement. DeviceType is an abstract type that inherits from TopologyElementType, and in the specification Device is taken to be any instance of a type inherited from DeviceType. BlockType is also abstract, and industry groups are meant to standardize general purpose BlockTypes. The purpose of ConfigurableObjectType is to provide a way to create modular topology elements, or in other words to provide means for Blocks or Modules to be organized under Devices. ProtocolType is used to represent communication protocols such as different fieldbus protocols which are implemented by certain TopologyElements. [11]

### 4.4.2 Analyser Devices information model

The Analyser Devices information model is a similar specification to the Devices information model, except that it deals specifically with analytical devices. Examples of analytical devices are light spectrometers, particle size monitoring systems and mass spectrometers, as well as compound analysers composed of several individual analysers. The four basic types of the Analyser Devices model are AnalyserDeviceType, AnalyserChannelType, AccessoryType and AccessorySlotType. They are shown in figure 13.

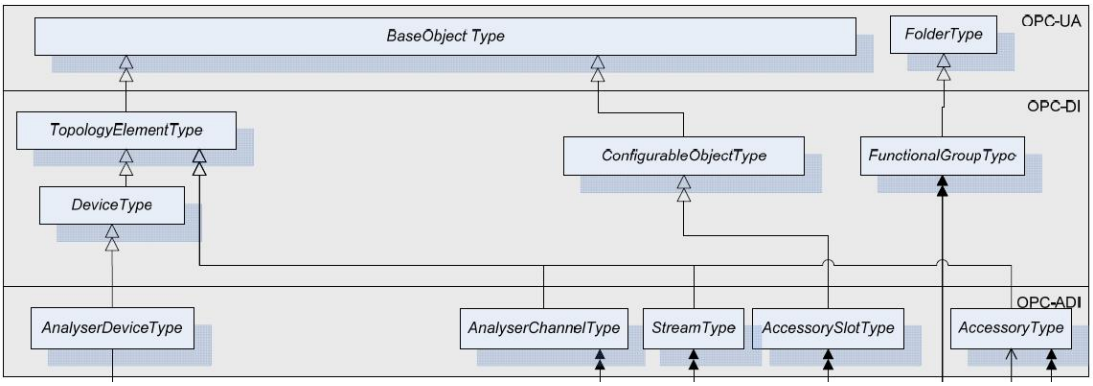


Figure 13: Analyser Devices types and relation to Devices.[12]

AnalyserDeviceType is used to represent the structure of an analyser, and is a subtype of DeviceType, defined in the Devices information model. Each analyser type can then be modelled as a subtype of AnalyserDeviceType, and six of these subtypes are introduced in the spefication. AnalyserChannelType defines the structure of AnalyserChannel objects, and is a subclass of BaseObjectType. AnalyserChannels have zero or more AccessorySlotTypes, which represent physical connection points to where analytical accessories can be attached. AccessorySlotType is a subtype of ConfigurableComponentInterface. AccessorySlots can have Accessories, whose type is AccessoryType which is a subclass of BaseObjectType. The Analyser Devices model also defines state machines for representing states and commands of the subclasses of the AnalyserDeviceType, AnalyserChannelType and AccessorySlotType.[12]

### 4.4.3 IEC 61131-3

An information model for IEC 61131-3 is under development by PLCopen and OPC Foundation[13]. The latest version at the time of writing was 0.09.1, and the release candidate goes by the name OPC UA For IEC 61131-3. Its purpose is to define an OPC UA information model to represent the models in IEC 61131-3. IEC 61131-3 is a global standard for industrial control programming. It standardizes programming languages to be used in industrial automation, and is independent of individual vendors.

The information model specification for IEC 61131-3 is based on the UA Devices information model. Figure 14 shows the type inheritance from UA via Devices to IEC 61131-3 OPC-UA.

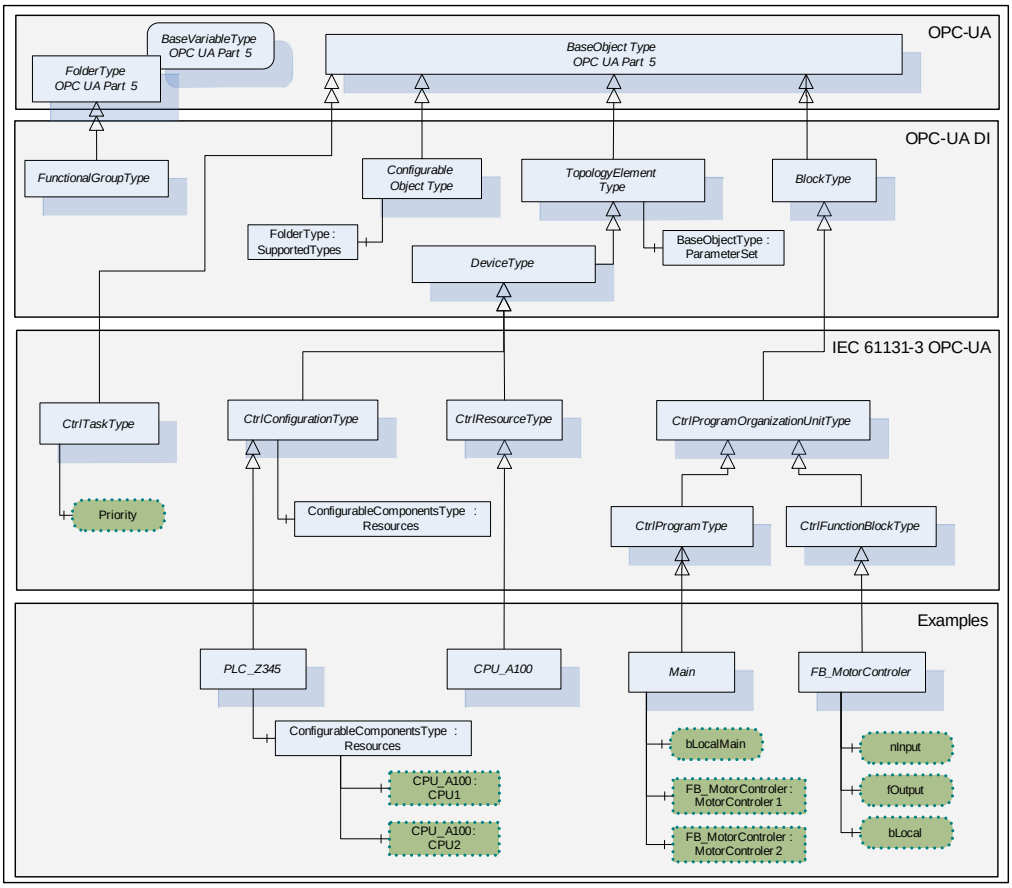


Figure 14: IEC 61131 3 ObjectTypes Overview.[13]

CtrlTaskType represents a control task of a programmable controller system in the OPC UA AddressSpace. CtrlConfigurationType represents the control configuration of a controller system, and CtrlResourceType is used to represent control resources of the system. CtrlProgramOrganizationUnitType is derived from Devices' BlockType, and it is an abstract type. It has two subtypes: CtrlProgramType which represents a program, and CtrlFunctionBlockType, which represents a function block.[13]

## 4.5 Already existing implicit models

### 4.5.1 General

In many legacy systems an information model already exists, just in an implicit form. For example, systems may have complex templates to represent certain devices, which can be seen as the type for these devices. In these cases, the task is to map

the implicit models into an OPC UA information model. Thanks to its flexibility, OPC UA can accommodate any information model. For example, the OPC UA wrappers provided by the OPC Foundation for the purpose of wrapping classic OPC servers for use with OPC UA clients just show the OPC server as an addition to the address space, using classic OPC as the information model. While the OPC information model is very flat, other kinds of systems might have hierarchies in them, either implicit or explicit. These models can then be made visible in the OPC UA Address Space and used in application. The next section, 4.5.2, describes one case of this kind.

### 4.5.2 Extending a process database with OPC UA

Currently OPC is used with Neste Jacobs's NAPCON TMLDB realtime process database[23] for connections to automation systems. The advantages seen in upgrading to UA from OPC are more flexibility, getting rid of DCOM communications, the possibilities of information models in transferring process knowledge and generally updating the system to more current technology[24]. The UA Address Space is considered to be very suitable for transferring existing database data[24]. There are several types defined in the database system, and they are available through OPC UA.

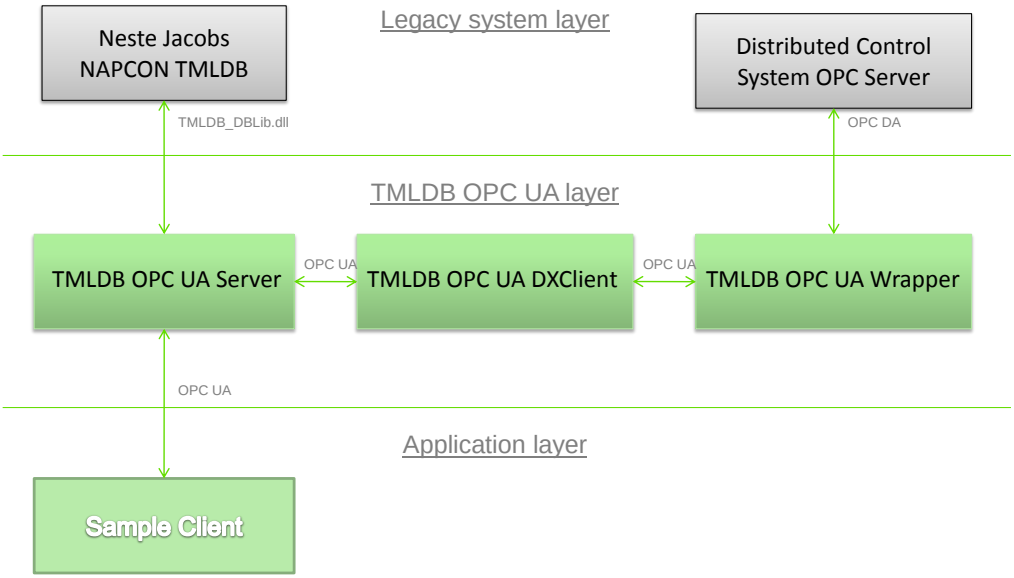


Figure 15: NAPCON TMLDB OPC UA demonstration.[24]

The first results were presented at OPC Day 2009, where a demonstration involving transfer between NAPCON TMLDB and a commonly known automation system

vendor's OPC server was presented. Figure 15 shows the layout of the demo.UA support was added to TMLDB using a dedicated OPC UA server which wraps the database and enables access to it, called TMLDB OPC UA Server. The metsoDNA OPC server was wrapped by a simple OPC <-> OPC UA Wrapper, called TMLDB OPC UA Wrapper. Accessing these two UA servers is a client, called TMLDB OPC UA DXClient, which subscribes to changes of a number of Nodes on each server and writes any changes to the other server. The Nodes to be listened to as well as the corresponding Nodes on the other server are defined in a file. The communications between the UA Server, the DXClient and the UA Wrapper are all done using OPC UA. Communications between the UA Wrapper and the DCS OPC server is done using OPC, so all the usual limitations of DCOM apply. Other clients can of course also access either one of the UA servers.

The TMLDB OPC UA server connects to TMLDB at startup and constructs its Address Space based on discovered data. While data groups, or tags, have type information, it is not used in the preliminary implementation, but is planned to be accessible in the address space later. A few sample types, XCV and PID, are presented next and UA types are sketched for them. This will serve as an example of how existing information models may be mapped to OPC UA information models.

Table 4: XCV type attributes in TMLDB.

| **Attribute** | **Data type** | **Description** |
|---|---|---|
| VALUE | Long | An enumerated value which tells the state of the valve. Can be closed, open, closing, opening or error. |
| LSO | Double | Used to open the valve. |
| LSC | Double | Used to close the valve. |
| OPERABLE | Long | An enumerated value telling whether the valve is operable. |
| TYPE | Long | When this is set to one, switching between automatic and manual control is possible. |
| MODE | Long | Whether the valve is on automatic or manual control. |
| POP | Long | Force opening the valve. |
| PCL | Long | Force closing the valve. |
| IOP | Long | Used to block opening the valve. |
| ICL | Long | Used to block closing the valve. |

XCV is a type which represents certain valves in the TMLDB system. The attributes of the type have to do with mainly opening and closing the valve and the state of the valve. The attributes of the XCV type have been presented in table 4, along with their data types and a brief description of each.[25] It is important to note that many of the variables which have Long as their datatype actually function like booleans, with 0 being false and 1 being true.

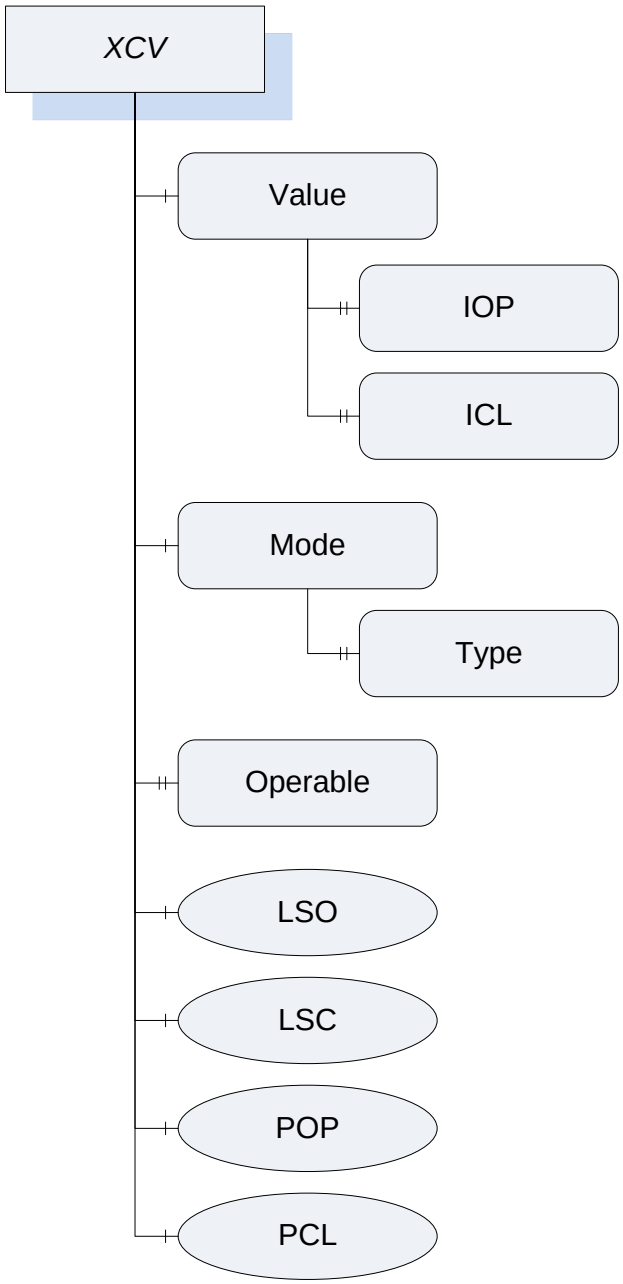


Figure 16: TMLDB XCV as an UA Type.

XCV is converted to an UA Type by first determining a NodeClass for each attribute, since each of the TMLDB attributes is represented by its own Node in the type. XCV will be represented by an ObjectType, and when instantiated in an AddressSpace, it is an Object. Value, Mode and Operable are Variables. IOP and ICL should be modelled as children of Value, since they dictate how it can be changed. The HasProperty ReferenceType should be used for this. The same goes for Type as a

property of Mode. Operable should be connected to XCV using the HasProperty ReferenceType, since it is directly related to XCV and has no children. LSO, LSC, POP and PCL are used to control the valve. For this reason, they should be modelled as Methods in the type. XCV as an UA Type is presented in figure 16.

Table 5: PID type attributes in TMLDB.

| **Attribute** | **Data type** | **Description** |
|---|---|---|
| SETPOINT | Double | The setpoint of the controller. |
| CONTROL | Double | The output of the controller. |
| GAIN | Double | The gain of the controller. |
| INT_TIME | Double | The integration time of the controller. |
| DER_TIME | Double | The derivation time of the controller. |
| MOUT_MIN | Double | Minimum limit for the output. |
| MOUT_MAX | Double | Maximum limit for the output. |
| MODE | Long | An enumerator for the mode of the controller. |
| TRACKING | Long | Whether tracking is on. |
| CONUNIT | Long | An enumerator for expressing the unit of the output of the controller. |
| A1 | Long | An alarm byte. |
| A2 | Long | An alarm byte. |
| XS | Long | A blocking byte. |
| ESD | Long | A protection byte. |
| R_ON | Long | The controller may be put into remote mode. |
| C_ON | Long | The controller may be put into computer mode. |

PID represents PID controllers, which are commonly used in industry. It has current values, tuning parameters, limits and modes. Table 5 presents the attributes of the PID type.[25]

PID itself is an ObjectType. The attributes deemed most important, setpoint, control and mode have been modeled as Variables and linked to PID with a HasComponent ReferenceType. An Object called TuningParameters serves as a way to organize the tuning parameters of the controller into one place. Gain, IntTime and DerTime are Variables and are linked to TuningParameters using the HasProperty ReferenceType. Control and Mode also have properties that are related to them, all linked with HasProperty. The blocking byte and the protection byte were deemed important enough to be modelled as properties of PID itself. In addition to this structure, all the Variables in the type have their UA DataType defined, using the HasTypeDefinition ReferenceType. The two alarm bytes are represented in the type as a Condition, referred to using the HasCondition reference type. In OPC UA, alarms would be events generated by the PIDAlarm, and separate alarm flags are contained within that Condition. Clients interested in the alarms would

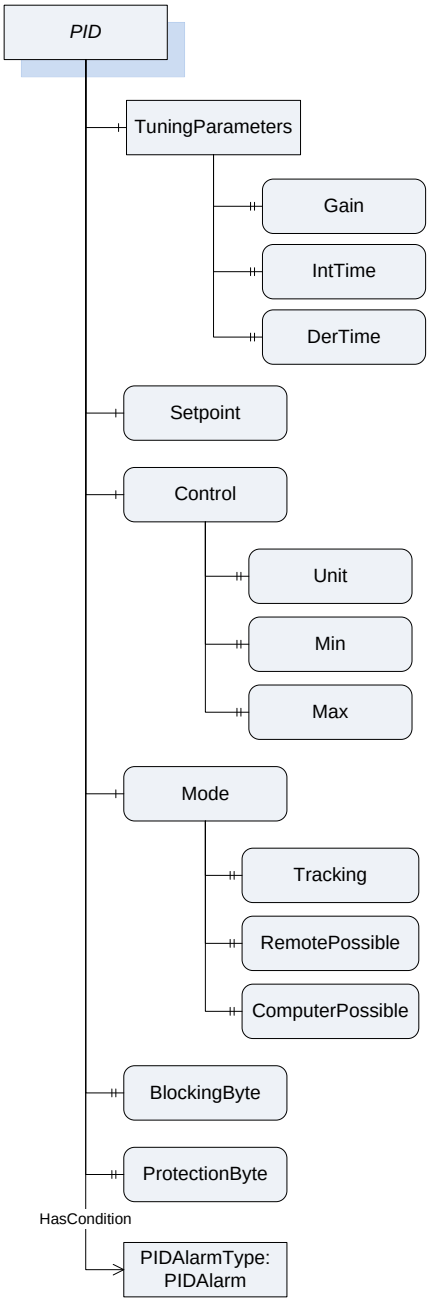


Figure 17: TMLDB PID as an UA Type.

either listen to events from the PID Object or sucscribe to the value of an alarm byte inside the Condition. In this example, the Condition type is not specified in full. PID as an UA Type is presented in figure 17.

The XCV and PID types demonstrate that while some aspects of existing information models may be mapped almost directly to OPC UA, some may be mapped to advanced mechanisms, such as Methods or events. This does not hinder interoper-

ability in any way, since the UA server may wrap the underlying implementation, whatever it may be, and present it as an OPC UA compliant model.

## 4.6 Semantic modelling

Since the information models are structured models laden with semantics specific to the application field, it is natural to consider them in the light of semantic models. The OPC UA Address Space Model[7] defines the basic elements that can be used in OPC UA to define objects and their relations. A basic data model used in semantic applications is RDF (Resource Description Framework).[1] It has many similarities with OPC UA which are worth noting. UA Nodes correspond to RDF resources, UA ReferenceTypes to RDF properties and UA References to RDF statements. In addition, all resources have an unique identifier, which corresponds to UA NodeIds. Because the OPC UA meta model essentially defines a directed graph structure which includes both types and instances, semantically rich information in ontology forms can be made available through OPC UA.[26] As Aarnio et al.[26] see the uses of OPC UA, "...it could be feasible to start building domain specific information models based on the corresponding standardized domain ontologies. In addition, standardized ontology language serializations could be used in model exchange and UA server Address Space configuration."

Semantic description languages vary in their expressive power, and a related issue is the ability of reasoning engines to work on the ontology expressed in a language, also known as decidability. The balance between expressiveness and decidability is a trade-off, to increase one other the other has to be decreased. An example of this are the different variants of OWL (Web Ontology Language). They are OWL Full, OWL DL and OWL Lite, in order of decreasing expressiveness and increasing decidability. While valid documents of each are also valid in RDF, the inverse applies only for OWL Full. Unfortunately, OWL Full shares RDF's weakness too, that being undecidability. OWL DL and Lite both limit the expressiveness of OWL Full, thus gaining decidability.[1] An interesting question regarding the semantic nature of the UA models and their general usefulness is whether or not the basic UA Address Space model is decidable, and if it's not, what sort of restrictions would make it decidable. Decidability is necessary for the use of reasoners, which could prove useful in future OPC UA applications. Determining whether or not it is decidable is a task well out of the scope of this thesis, but nevertheless an interesting question which has a large impact on future applications of UA information models in conjunction with reasoners.

One way of furthering interoperability in OPC UA based systems would be to use certain standardized types in the information models. This way, the semantics of the information model would be based on an industry standard. The RDS/WIP system[27] is a type library which could be relevant for this purpose. It is a SPARQL-enabled (SPARQL Protocol and RDF Query Language) library based on the ISO 15926 standard. ISO 15926 is a standard developed to enable integration of life-cycle data, originally in process plants.[28] Most interesting in the scope of this

thesis is Part 4, which defines application- or discipline-specific terminologies.[28] This vocabulary is what is presented in the RDS/WIP system online and thus can be programmatically accessed using SPARQL queries. This kind of standardized vocabulary could serve as a basis for types in OPC UA information models. For example, specific boiler types could be subtypes (using the HasSubtype reference) of an abstract boiler type, which is specified in the RDS/WIP system. This way, different systems would have a common conception of exactly what a boiler is, thus enabling the development of more intelligent applications.

# 5 Use of information models in applications

This section details the use of information models in applications, starting from different use cases and ending with the implementation of the applications that use the information models.

## 5.1 General

Information models are defined to be used in applications, and so must serve some purpose regarding the application field. Handling information models can be implemented in various ways in the server software, each with its own benefits and limitations. In addition, clients can use the information model provided by the server in various ways. Most often, the client will be aware of the information model and will be designed to utilize it. However, since the model is also explicitly presented in the server as types, all clients can access data whether or not they are previously aware of the information models on the server. It is also noteworthy to consider how the server is bound to the actual data source, such as an automation system etc.

## 5.2 Use case examples

This section first discusses some possible use cases from the perspective of design and from the perspective of runtime flexibility, and then presents a few possible use cases for server support of information models. Real applications might have to deal with several overlapping use cases. The use cases are listed in order of increasing need for flexibility from the server implementing these models. This section and the next, 5.3, try to identify use cases and possible implementation strategies, while 5.6 tries to sketch a rough pairing between them. The accuracy of the use cases and implementations here as well as their frequency could warrant further research.

### 5.2.1 Design time use cases

Information models may be defined in software by two different methods: either hand-coding them in the software or using a design tool to define them. Design tools may either generate code or create some sort of intermediate file, such as an XML file. In either case, it is assumed here that the information model itself has been created and the task is to take it into use in a server.

**Hand coding** is the most straightforward method of designing an information model. Using this method, all types and instances are defined by hand by writing definition files or code files by hand. Possible bindings to underlying data systems are also hard coded. This may be the design use case for very

small servers, such as embedded servers, where it is not necessary to configure a large Address Space and where overheads must be kept to a minimum. Another use for hand coding are certain internal tasks that the server must handle, such as providing its own status in the AddressSpace. Hand coding will require some pre-defined elements, such as type identifiers, to be easily available in the IDE (Integrated Development Environment) used in coding.

**Design tools** can be used to create larger Address Spaces, and may also be useful when combining several different levels of information models. Some tools already exist (presented in this thesis in section 5.5), and it can be expected that more will appear on the market in the future. Designing information models using a design tool is very different from implementing them directly in code, but it also requires an additional framework for the deployment of the designed model on the server. Current tools rely on code generation from the designed model, but the implementation made in this thesis provides one alternative approach, adding the models to the server dynamically at startup. How the data binding to the underlying system should be made is left to be coded by hand in the existing implementations, and a simple example of dynamic code binding is demonstrated in this thesis.

### 5.2.2 Runtime use cases

Servers may or may not require changes to information models, bindings to the underlying system etc. during the server running. Depending on the server, it may not be possible to shut it down to replace code. Two different use case scenarios are presented next.

**Static address spaces** may be found in, for example, cases where a OPC UA server wraps some legacy system, such as in the cases described in section 4.5. It can be thought that most of the time, these kinds of models are somewhat static and thus do not require changes in the server's address apace. An example would be a field bus with sensors and actuators plugged in, whose structure is likely to remain unchanged for long periods of time. Another example would be that the underlying system consists of several parts which may change, although not very often and at predictable times. These kinds of systems do not require the ability to change types etc. at runtime. If necessary it is possible to bring the server down for maintenance and update the address space.

**Dynamic address spaces** mean the situation when it is necessary for the address space to change often and types are modified, removed and added frequently. The server needs to be able to handle these changes while operating normally otherwise, in other words no changes which cannot be done at runtime can be done. This is also true if the server configuration is made via a configuration client, meaning that the server is up and running when the information models

are actually added there via OPC UA NodeManagement Services. Remote management of OPC UA servers will also benefit from the possibility to make changes while the server is running. An example of a server which would require a dynamic address space would be an CRM system with different types for each customer's order formats. The server cannot be brought down every time a customer is added to the system.

Generally speaking, a static implementation may be easier and cheaper, especially with small models. If changes are needed during the lifetime of a server, the benefits of a dynamic solution outweigh the initial costs.

## 5.3 Implementation of information models in the software

Support for different infomation models can be implemented in a piece of software in several ways. Although the abstract meta model of OPC UA represents information as Nodes, the actual software implementation can use some other way of handling information. The most obvious alternative to using actual software objects to represent Nodes is to have the infomation in a relational database, which is then wrapped to be accessible through OPC UA. This thesis does not cover the creation or use of information models in database-based implementations, and the following implementation choices are based on the assumption of an object-based Address Space implementation.

Depending on the application, different ways to implement the information models can be advantageous. For example, the information model for an elevator will stay the same indefinitely, and thus there is no reason not to hard code the information model in the application. Hard-coding information models is relatively if using an IDE, if the SDK which is used has suitable pre-defined classes to support it. IDEs can provide autocompletion and other helpful functions to assist in defining the information model in code. On the other hand, a server which faces frequent model changes needs to be able to handle the model very dynamically, and thus using generic classes to implement the model will be the best choice. It is noteworthy that these different approaches are not mutually exclusive and can be used side-to-side in suitably designed server software, so that the server will be able to support whichever method is the best for the situation. Runtime techniques allow more varied changes to the address space, but they are more difficult to implement. Defining models is also more complicated if modelling is done in runtime and will require some kind of designer software.

The following lists different approaches in order of increasing dynamicity at runtime and decreasing ease of deployment.

**Hand writing** removes the need for any tools for the operation. While very straightforward, this approach is not well-suited to anything but small information models. Convenient use of this technique also requires using an IDE and an SDK that has suitable classes to support model definition.

**Code generation** is a way of defining the information model at compile-time. In this technique, the source code of the program will be partially auto-generated to create information models. The result is that there will be object classes representing each node type. A possible downside to this implementation is that addition or modification of information will require regenerating the source code and rebuilding the application, which is not convenient if it is to be expected that the information model changes often. Because of their static nature, this approach is well-suited creating the standard OPC UA information model on the server. This is the approach used in the .NET implementation provided by the OPC Foundation, as well as in Unified Automation's C++ SDK. In the .NET implementation there is a hierarchy of abstract superclasses which represents the basic UA Node types, and specific classes inherit these to top off the hierarchy. For example, in the samples provided with the SDK, there is a ValveState class which inherits GenericActuatorState (another generated class), which in turn inherits BaseObjectState, a non-computer generated class.

In addition, it is possible to use dynamic classes, which means that an application might add or modify source files during execution. This allows a running application to create necessary classes for new information models and load them at runtime, removing the need for recompilation. However this method would be more complicated and thus costlier to implement than simply generating necessary classes.

**Generic node classes** can remove the need to do structural modifications to the program itself. The basic idea is that each NodeClass should have its own object class, so that the object model in the server and the abstract UA version of it are the same. The whole AddressSpace is built in the server using just eight object classes, which are flexible enough to be customized to the specifications of the information model. Customized behavior can be added by registering custom code handlers to Nodes. This is the approach taken in the solution developed in this thesis, and the solution also has a sample implementation of adding listeners to bind custom code to Nodes.

It is important to note that ideally all three of the above could be used side by side in development with minimal overheads, and the one best suited to each individual use case used for that one.

## 5.4 Implementation of custom code handling in the software

In many cases, information processing can be added to an UA server by using a client which accesses the server, performs the necessary operations and stores the results in the server. However, this may not always be the case. Reasons for running code in the server itself include modifying access to Nodes, execution speed, and of course UA Methods.

Modifying access means that client access to the Nodes and their information must be processed in some special way. It may be that only certain values are permitted to be written to an item, that an item may not have an actual value but is dynamically determined from other items upon request or that the server needs to access some underlying system in order to accomplish the client's request.

Another reason for running code in the server is that in time-critical applications delays caused by communication errors, lags, or other disturbances must not endanger the execution of the code.

A Method is in the context of OPC UA "a callable software function that is a component of an Object."[6]. They are visible in the Address Space as components of Objects and clients use the Call service to invoke them with input arguments, and get output arguments as response.[8] It may be that the server performs some tasks upon invocation of a Method or it may be that some underlying system is called upon. In both cases, custom code is necessary to define what exactly should happen when a Method is invoked.

The running of custom code may have several kinds of triggers, three of which are presented next.

**Execution on access** means that the code is run whenever a client accesses a certain Node in the server. For example, if the server wraps a fieldbus, it may be that whenever a client reads a value, it needs to be retrieved from the fieldbus and then passed on to the client.

**Execution on request** is the case with Methods. When a Method is called, the server needs to find the code the Method is linked to and run it. Using the previous fieldbus example, if a pump Object in the AddressSpace has a Method defined to start it, then calling this method should cause the server to signal the pump to start via the fieldbus.

**Periodic execution** has to do with various recurring tasks, such as calculations, server maintenance, etc. It may be that a Variable in the AddressSpace represents the moving average of a certain other Variable, and it should be recalculated every ten seconds.

Each of the three code activation methods has its own uses. The solution in this thesis will present a sample implementation of the 'Execution on access' case, where code is run when client accesses a Node without the client specifically requesting it.

There are also different alternatives in how the custom code and the AddressSpace interact. It may be that the server signals different processes in the server machine or other machines to start, or that the custom code actively uses the AddressSpace during the execution, fetching parameters etc. Three alternatives are presented below.

**Code running beside the Nodes** refers to a case where the code can access the AddressSpace, to retrieve parameters kept there for example. This is a

lightweight solution for use cases where there is no need for an extensive backing system, and making such a system just consume extra time. Code can use information in the server's AddressSpace while it's running, so there is no need to couple the same information to any backing system.

**Code running behind a facade of Nodes** is the case where the code runs in the server process, but is self-sufficient and doesn't require access to the AddressSpace. There is a backing system that is in the same process as the server, and it may even be that the OPC UA server is just a component of a larger application whose information it presents via OPC UA. Service calls are translated from OPC UA to whatever the underlying system requires, and the server acts as a facade for the rest of the program.

**Code running in the background** means that the code is in different processes which are simply controlled via the server. The binding needs to specify how to control the other processes when called for. An example would be a DCS running in the same machine as the OPC UA server. Then the OPC UA server acts as a wrapper for the DCS, translating between the propietary interface that the DCS provides and the interoperable interface that the OPC UA server provides.

Of course, the exact code that needs to be run in an application is to be decided by the application developers, but it is beneficial if a SDK offers a solid base for writing the code. Code may be bound based on the namespace, the type, or even for individual Nodes as the need dictates.

## 5.5 Existing information model tool implementations

The first two implementations listed here are based on the code generation technique presented in section 5.3, so they generate code that will define the information model in the application. The third, OPC UA Address Space Model Designer, generates XML files. Simantics is work in progress and how models .

### 5.5.1 .NET Model Compiler

The .NET SDK has a code generator with it, Model Compiler.[3] It takes as input an XML file containing node definitions and another file, in .csv (comma separated values) format containing NodeIds for the nodes. Based on these two documents it generates the necessary code for the information model to be used with the .NET SDK. The generated code is based on code templates, and the classes generated by the tool are partial classes, so the generated code may be augmented by hand. This means that special functionality can be written in by hand while still using code generation to keep the base class up to date with possible modifications to the model design. Model Compiler has no other functionality apart from converting

information models defined in XML files following a certain schema to code which is .NET SDK compatible and can thus be used with it. It has no editing capabilities and the actual creation of the information model XML has to be done using some other tool, or by hand.

### 5.5.2 UAModeler by Unified Automation

Unified Automation has developed an information model creation and code generation tool called UAModeler, which is customizable with regard to its inputs and outputs.[30] This approach, like the .NET SDK implementation, creates code to be included in an application at compile-time. The code generated is based on templates, and at the time of this writing only templates compatible with the Unified Automation C++ SDK were included in the tool. However, new templates to cover any use case or language can be created for the tool if the full version of the tool is in use.[29] Input to the UAModeler can be any XML file (with suitably defined filters) and in addition, the tool has a GUI interface for creating and editing information models.[29] It is presented in figure 18.

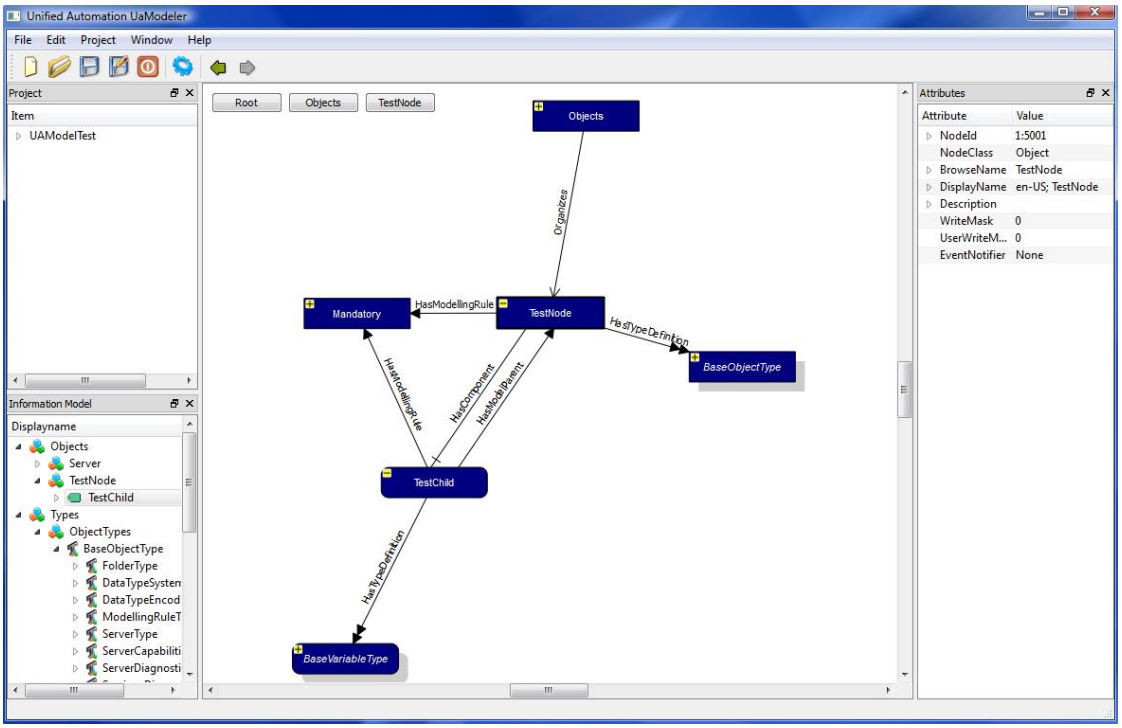


Figure 18: UAModeler GUI.

The GUI interface offers a node graph visualisation of the model, which can help understand the various references between nodes, but on the other hand, can be confusing at times. Creation of new nodes and references is simple, although at the time of this writing the tool seems still lacks some finishing touches, like that namespaces are identified by an index instead of their full URI (Uniform Resource

Identifier, an unique string which identifies a resource). Overall the UAModeler is a good tool for working with information models, but since there aren't any Java templates available for it at the time of this writing, it hasn't been used in this thesis.

### 5.5.3 OPC UA Address Space Model Designer by CAS

CAS has also developed a tool for designing UA information models, called OPC UA Address Space Model Designer[31]. The evaluation version of the tool features a basic way of creating and editing information models graphically, and the tool can generate XML which can be used with the .NET code generator. A screenshot is presented in figure 19.

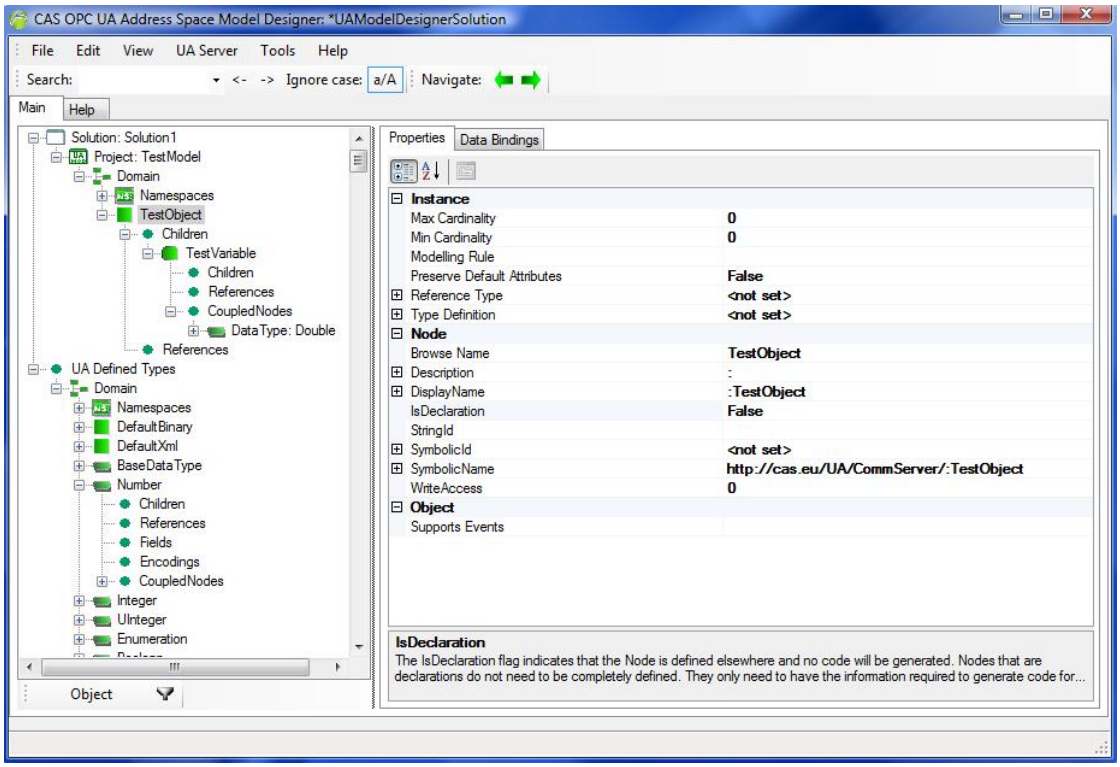


Figure 19: CAS Designer GUI.

The tool is not as easy and intuitive to use as UAModeler, but for creating information models for .NET it is a workable solution. Also, because the solution developed for this thesis relies on the same information model definition schema (which is detailed more closely in section 6.3.3), the CAS UA Model Designer can also be used to define models for this solution.

### 5.5.4 Simantics

The Simantics modelling and simulation platform[32] is a work in progress at VTT. It is defined on the Simantics website as "a software platform for modelling and simulation. The system has a client-server architecture with an ontology based modelling kernel and Eclipse framework based client software with plug-in interface." It has an ontology editor for creating and editing semantic models. Simantics would be well suited to creating OPC UA information models, and it will support them at some point in the future. The transfer of models from Simantics to UA servers could be done either by using intermediate files, or integrating a UA client into Simantics so that the models could be transferred using the NodeManagementServiceSet.

## 5.6 Synopsis

Information model instantiation and code binding can be done in several different ways, and the choice depends on the use case. Whether or not the server's AddressSpace will change a lot is a large issue, since hard coding models into an application is not well suited for a server into which models and Nodes are added frequently. Another important factor which should affect the chosen implementation method is whether or not a tool is available to use when defining the model. If an existing tool cannot be used, one may be developed, which will produce development overhead that might not be acceptable except in larger applications.

The use cases and implementation possibilities are shown in Figure 20, along with the positive and negative sides of each solution.

In the matrix, the top left cell is the one with the least overhead, but also with the least potential for large and changing information models. The solutions are not mutually exclusive, hybrids may also be used, but the overheads will cumulate if several solutions are used. Of course, if a suitable SDK that supports all choices and a design tool are available, there is no additional development overhead in using more dynamic solutions.

In addition to choosing a method for designing and instantiating the AddressSpace and its information models, one or more of the code binding methodologies described in 5.4 may be employed to add functionality to the server.

Use cases

Implementation choices

| | Hand coding, static address space | Design tool, static address space | Hand coding, dynamic address space | Design tool, dynamic address space |
|---|---|---|---|---|
| Hand writing | The code defining a model is written by hand to define a static address space.<br><br>+ Easy to implement, little overhead<br>- Unsuitable for anything but very small models | Not applicable | Not applicable | Not applicable |
| Code generation | Model code is generated from a hand written definition using templates.<br><br>+ Code generator allows larger AddressSpaces<br>- Hand writing of definition files prone to errors<br>- Server will require recompilation if new models are added | Model code is generated from a designer generated definition using templates.<br><br>+ Design tool easier to use than hand writing<br>- Server will require recompilation if new models are added | Not applicable | Not applicable |
| Generic classes | Hand written model definition files are loaded at startup to create an address space.<br><br>- Generic classes would not be necessary, since a static implementation would have sufficed<br>- Hand writing of definition files prone to errors | Designer generated model definition files are loaded at startup to create an address space.<br><br>- Generic classes would not be necessary, since a static implementation would have sufficed | Hand written model definition files are loaded at startup or during runtime to create an address space.<br><br>+ Dynamic implementation that is modifiable during runtime<br>- Hand writing of definition files prone to errors | Designer generated model definition files are loaded at startup or during runtime to create an address space.<br><br>+ Dynamic implementation that is modifiable during runtime<br>- Both generic classes and a design tool are needed |

Figure 20: Use cases versus implementation options.

# 6 Solution

This section details the object-oriented information model implementation made in Java.

## 6.1 Introduction

With one of the design goals of the SDK being that it can be used for any kind of application development, it is important that the underlying information model implementation can handle any information model and a multitude of use cases. The implementation needs to be modular, so that different input methods and NodeManager implementations are supported. On the other hand, while the normal use case would be to export information models from an XML, it should be also possible to modify the information model from another application. A natural choice for this use case is to use the inbuilt NodeManagement services in OPC UA, so that any application that wants to modify the AddressSpace can do so using just OPC UA. This approach is also very much in the spirit of UA, and supports the goal of modularity.

## 6.2 Design goals

The solution will be developed as an information model handling framework in the Java Sample Server. The developed framework needs to support information models that are loaded at startup as well as the possibility to modify the AddressSpace at runtime via NodeManagementServiceSet. Use of several different NodeManagers for different information models should be possible. The namespace of each information model should dictate which NodeManager it will be added to.

A sample NodeManager which allows adding custom code to Nodes is a part of the solution. The NodeManager will use the Nodes' type information to determine what code should be linked to the Nodes. A simple example will be made and presented to serve as a proof of concept.

## 6.3 Existing platform

### 6.3.1 Overview

This section will present AddressSpace implementation of the Java Sample Server, which serves as the base for the solution presented here. Also, the information model definition format used in this thesis is explored in more detail.

During the development of the Java Sample Server, direct hand-coded instantiation of information models has been used. In other words, there have been no automated tools for deploying information models. Also in some parts of the Sample Server

support for using several NodeManagers has been incomplete. However, these have been rectified during the development of this solution.

### 6.3.2 Address space implementation

The implementation of the address space in the Java Sample Server is very similar to the one in the .NET stack. The nodes themselves are stored in a system of node managers. This includes both instance and type nodes. In addition, type nodes are held in a TypeTree which allows for efficient operations on their inheritance relations.

The system of NodeManagers is shown in figure 21. The Server object has a reference to a MasterNodeManager, which receives AddressSpace related tasks and delegates them on to the actual NodeManagers. For example, when the server receives a Read service call, it passes it on to the MasterNodeManager. The MasterNodeManager gives the list of nodes to be read to each NodeManager in turn, and for each node the owning NodeManager reads the node. Once all NodeManagers have been called with the list, all the read requests have been processed.

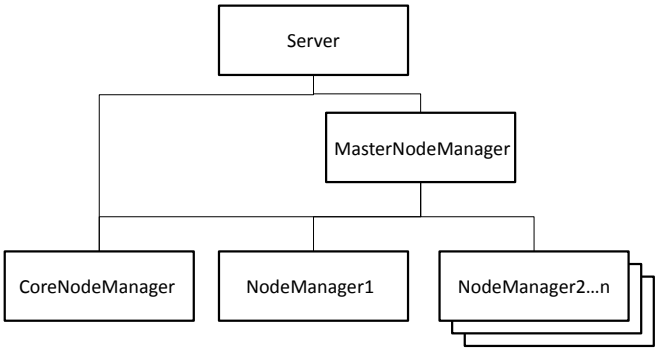


Figure 21: The hierarchy of NodeManagers.

Each node manager handles a part of the total address space. Reasons for creating custom node managers have to do with inserting custom behavior into how the nodes are handled, for example one node manager could be an interface to a field bus and another an interface to a database system. From the perspective of OPC UA, there is no need to actually have node objects that hold information. It is up to the node manager how it implements its interface. However, if there are no special needs regarding underlying systems, custom node management behavior, etc., there is a default node manager called CoreNodeManager available. It is based on an object

implementation of the nodes and can be used to handle nodes which have no special needs. The types of the server reside in the CoreNodeManager, as do the nodes related to the server status.

### 6.3.3 Information model definitions

The idea in this solution has been that the information models which are to be added to the server are defined in a file, which the server can read at startup. In the Java Sample Server, the base AddressSpace that is common to all UA servers is read from a special XML document using the Xstream XML serialization/deserialization library[33]. This XML format lacks a schema and is different from the one used in the .NET implementation to represent information models. The schema used in the .NET implementation for information models, UA Model Design.xsd[3], will be used in implementing the XML import. The schema is presented in its entirety in Appendix A. The creation of the XML documents adhering to this schema is unfortunately beyond the scope of this thesis. There are a few tools which may be used to create such models, which have been presented in section 5.5. For the purposes of this thesis, it is assumed that the XML already exists. This is a reasonable assumption, as for small models the XML file may also be written by hand, as has been done for the example in this thesis.

XML files following the UA Model Design.xsd are relatively simple and comprehensive. In essence, the file contains a list of Nodes as elements, and the node attributes are expressed as either XML attributes or child elements. A small example of a model definition is found later with the example in section 6.5.2. One hindrance the schema suffers from is the close link between the .NET implementation and itself. This means that in addition to node attributes defined by OPC UA, the schema also includes information about how the .NET server should implement the information model. This extra information is specific to the .NET implementation and so is skipped in the Java implementation. Nevertheless, the XML files will provide all necessary information for the creation of information models.

One very important difference between the Java implementation and the .NET implementation is the handling of the children of nodes. In the .NET SDK, each node has children and references. A child is seen as being owned by the parent, so that if the parent node disappears, the children will too. References are links between independent nodes[34]. In the OPC UA specification, no such division between children and references is made, and all links between nodes are simply references. This is also the approach taken in the node objects in the Java stack. However, the distinction is important when the address space is modified. The children of a node are linked to its existence, i.e. if the parent node is deleted, so will the children. On the other hand, simple references do not have this kind of information embedded in them. The parent-child information in this sense is additional information that can be used in maintaining the address space. In the .NET implementation, the children are stored separately from other references in the parent, as an array. The reference type for these parent-child references is stored in each child node in

a ReferenceTypeId field, which defined the type for its parent's reference to itself. Another design choice that would achieve the same things would be to bundle all references, like in the specification, but keep a separate list of each node's children. This is what will be done in this implementation, as this list of children doesn't have to be a part of the node object itself. In this solution, the list of children is kept in the parts that manage the address space, and not in the NodeManagers themselves.

## 6.4 Framework implementation

This section will document the architecture of the solution.

### 6.4.1 Overview

The solution has three main parts. Two of the parts are interchangeable, to allow customized implementations. The first part is a component that handles the reading of input sources, and in this solution it will be implemented as an XML reader called ModelParser. This can also be any other component which needs to access and modify the Address Space of a server, for example an UA configuration client which accesses the server via OPC UA. The second component is the AddressSpaceHandler which will implement the NodeManagement Service Set which has been previously detailed in section 2.3.3 and manage the NodeManagers. The third component will be a specialized NodeManager called CustomObjectManager which will support custom code in the Nodes. Its internal implementation relies on node objects, so it is similar to the CoreNodeManager in most respects. The interfaces between the components have been formally defined, so that new implementations of particularly the input sources and node managers are straightforward to develop and deploy. Figure 22 presents the classes in the solution.

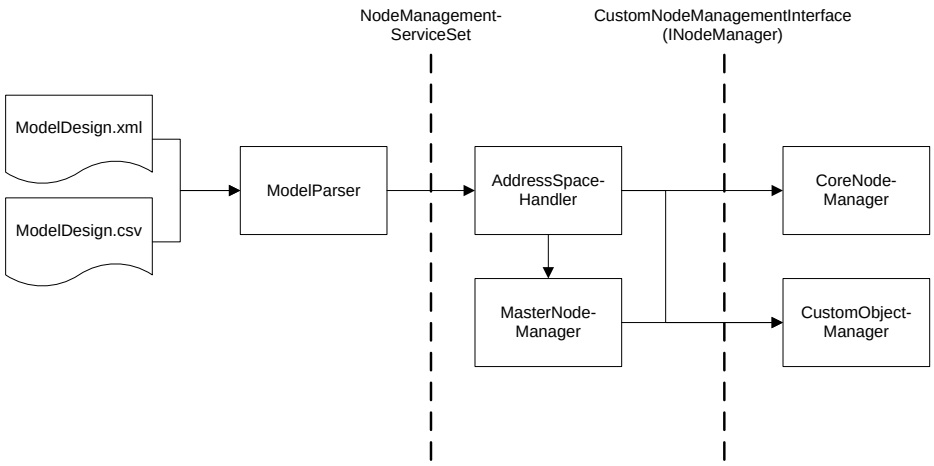


Figure 22: The solution.

Two vertical dashed lines in the picture reflect the way the solution has been modularized into three parts, input (in this case, the ModelParser), AddressSpaceHandler and NodeManagers. The line on the left represents the NodeManagementServiceSet interface, and the one on the right the CustomNodeManagementInterface.

In this solution, the Java interface that represents the NodeManagementServiceSet and which the AddressSpaceHandler implements is called AddressSpaceHandling. It is presented in Appendix B. In addition to four methods which correspond to the four different Services in the NodeManagementServiceSet, the interface also has two additional methods. registerChildren() is meant to be used by the ModelParser to inform the AddressSpaceHandler of the nodes' parent-child relations discovered during parsing. Another method, registerNamespaceManager() is meant to be used by custom NodeManagers to inform the AddressSpaceHandler of the namespaces they will govern.

CustomNodeManagerInterface is an interface the custom NodeManagers will have to implement, and it is a combination of three interfaces, INodeManager, NodeManagement and ReferenceManagement. INodeManager is an interface in the Java Sample Server that all NodeManagers, including the CoreNodeManager, implement. It has methods for normal operations with NodeManagers, like reading and writing. NodeManagement and ReferenceManagement have been defined in this solution to allow custom NodeManagers to support adding and deleting of either nodes, references or both. Both interfaces have two methods, so combined they correspond to the Services in the NodeManagementServiceSet. The service calls will be forwarded to the NodeManagers using these interfaces. This is because the interface of the NodeManagers will have to accommodate other address space implementations as well, especially as database implementations. No assumptions about the internal implemention of NodeManagers can then be made, and the obvious choice is to simply forward the service calls.. Although a database-based implementation is out of the scope of this thesis, the design choices made now must be flexible enough to allow such an addition in the future.

### 6.4.2 XML deserializer

One of the main goals in the making of this thesis is to build a working solution to import information models defined as XML-files to a Java server. The XML files will follow the schema put forth in the .NET stack, and in addition a .csv file will hold NodeIds for the nodes, which must be also integrated with the model. In considering how the XML-import would be handled, three separate implementation strategies were identified.

**A custom XML parser** built on either standard Java parsers or third-party parsers with a suitable license would be created. It would parse the XML file and translate the information into OPC UA NodeManagementServiceSet calls. Using a SAX (Simple API for XML) serial access parser would be suitable even for large information model files with instances defined. Its alternative, a DOM

(Document Object Model) parser would not be as well suited for the task. Reasons that support SAX are its efficiency with large documents, the fact that the information model is not monolithic but made of separate nodes that can (with a few exceptions) be read sequentially, and that it is not necessary to modify the document, just read it.[35]

**Using a serialization/deserialization framework** such as Castor[36], jaxb[37] or JiXB[38] the XML could be deserialized straight to the objects. The idea in the frameworks is that they map XML to Java classes and back, either automatically or using a mapping file. Regarding the Java implementation, there are four drawbacks to using them. First, they require the Java classes to be "bean-like" in their design (meaning be serializable, have a no-argument constructor and have properties that are used via setters and getters), which is not true in the case of the object classes used here. This limitation could be circumvented using factory classes to create the Java objects, but the factory classes would be another layer of complexity. Second, the definition schema, originally developed for the .NET implementation, does not map directly to the Java node classes that are going to be used here. This means that the mapping files would be long and complex, and dependent of the address space implementation. Third, the idea of dividing the implementation of the solution into three parts would not work, since the deserializer would need to be linked directly to the node object classes, and so using another address space implementation altogether (like a database) would require a different deserializer. Fourth, this would make it impossible or unfeasibly difficult to use custom-made classes for some nodes, as the binding file would need to be modified to take these into account. For these reasons, it was decided that using direct deserialization was not the best alternative, with bad reusability of the developed components and lack of support for custom classes being the main reasons.

**Preprocessing the XML** with an XSL Transformation (abbreviated XSLT) is another possibility. In this alternative, the idea is that a suitable XSL Transformation would be used to change the XML definition into a more suitable format before processing it. For example, the fact that the parent-child relationship is represented as structure in the XML, could be translated into a suitable element instead. This approach would have the advantage that the actual data collecting component which gathers the node information could be generic, and so different schemas (if necessary) would only require a new XSLT to be defined. This approach would also necessitate implementing one of the previous solutions, but would possibly simplify them and make them more portable.

From these three alternatives, the first one was selected. An XML parser that reads the model XML and .csv files and creates NodeManagementServiceSet calls from it was designed and implemented. The main reason for using this approach over a serialization framework was the need to decouple the address space implementation

from the model representation. Using this approach means that the model may be also uploaded through OPC UA, using the same services as the parser is currently using. Also, any preprocessing was deemed to be unnecessary, since the structure of the parser was found to be relatively straightforward.

The AddressSpaceHandler has a method loadNodesFromXml() which can be called to read custom information models into the server's AddressSpace. The AddressSpaceHandler instantiates a ModelParser, giving the name of the of the XML model to be loaded and the server's namespace table as inputs. Providing the namespace table simplifies the parser significantly, since new namespaces that are found in the XML can be added to it immediately instead of using temporary values. Before parsing the XML, the ModelParser reads the associated .csv file for the NodeIds, and places them in a temporary data structure. The parser acts in an event-based manner, which means it provides a callback interface to the SAX parser. When elements start or end, the parser extracts information and places it into temporary data structures. The ModelParser first loads the model from the XML and caches it, because it is possible that the information model in the XML is somehow malformed or invalid, which the SAX parser will not know until it reaches the invalid point. By using a cache, a model can be loaded into the server after it is certain the model is well-formed. Once the whole document has been processed this way, the AddressSpaceHandler, if the parsing has been successful, calls the createModel() method on the ModelParser. The ModelParser will then call the AddressSpaceHandler NodeManagementServiceSet calls, which results the information model contained in the XML to be added to the server. It could be thought that the ModelParser's task is really to convert the XML into NodeManagementServiceSet calls. Note that the AddressSpaceHandler has no information as to where the service calls originate, which means that the parser could be located in an OPC UA client, as the case may be with configuration clients. A sequence diagram for the addition of a model is presented in figure 23.

### 6.4.3 AddressSpaceHandler

The AddressSpaceHandler is the component that receives service calls in the NodeManagementServiceSet from the ModelParser (and from clients via OPC UA, etc.) and which forwards the instructions to the relevant NodeManagers. The AddressSpaceHandler implements the interface AddressSpaceHandling.

It is necessary for the AddressSpaceHandler to know which namespaces each NodeManager governs. It is assumed in this implementation that every node in a certain namespace will be handled by a single NodeManager, though no restrictions on the number of namespaces a NodeManager may handle are imposed. This is because each customized NodeManager is thought to have some special functionality, and the Nodes which rely on that functionality should then reside in their own namespace. Each NodeManager calls the method registerNamespaceManager() at startup with a list of all the namespaces it will govern. The AddressSpaceHandler takes note of these registrations and when a service call arrives, it can look up the NodeManager

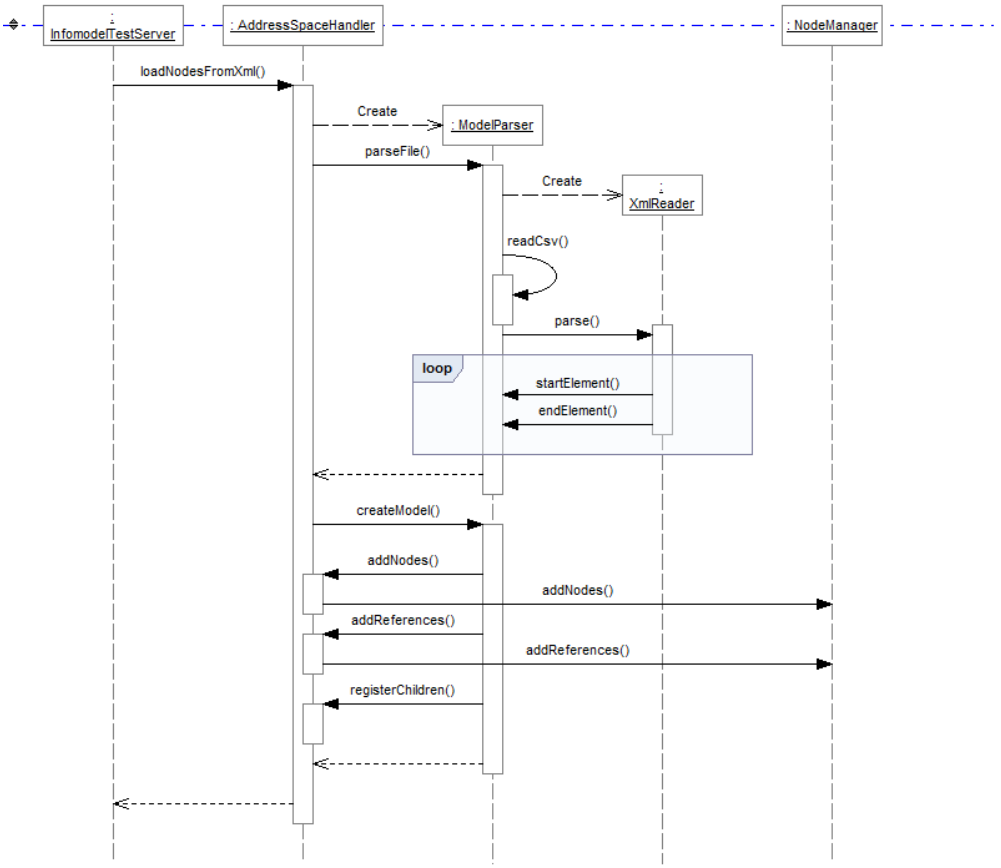


Figure 23: A sequence diagram showing the control flow in instantiating an information model.

to be used based on the namespace specified in the service call and pass the call on. The CoreNodeManager is a special case, as the service calls are not forwarded to it as such. For example, if a Node is to be added to the CoreNodeManager, the AddressSpaceHandler will construct the object and use MasterNodeManager's addNodeSet(), which will place the Nodes into the CoreNodeManager. The AddressSpaceHandler is thus tied to the implementations of the MasterNodeManager and the CoreNodeManager as well as the node objects used in them.

Not all NodeManagers can be assumed to support the operations of the NodeManagementServiceSet. For this reason, it was split into the NodeManagement and ReferenceManagement interfaces. The AddressSpaceHandler uses reflection to figure out which interfaces a certain NodeManager supports, and takes note of it for future use. In this implementation, the information is not used as the CustomObjectManager implements CustomNodeManagerInterface and thus both NodeManagement and ReferenceManagement, but this could be useful in future applications.

### 6.4.4 CustomObjectManager

A custom NodeManager that supports dynamic code binding to address space nodes is the last part of the implementation. The idea is to demonstrate that the AddressSpaceHandler can manage several NodeManagers and also to make a prototype of the generic class implementation detailed in section 5.3. The CustomObjectManager uses the basic Node object classes from the Java stack and allows type based custom code binding to them. Because of this, it shares most its code with CoreNodeManager, except for the code binding system.

The CustomNodeManager relies on the builtin feature of the Java stack Nodes that allows them to contain a handle for custom read and write operations. The handle is a simple object which contains references to both the node and the handler, which must implement an interface called IDataSource. IDataSource has two methods, for read and for write, so the data source which contains custom code can only react to those calls. This is somewhat more limited than the implementation in the .NET SDK where handler delegates can be registered for Nodes for several different operations. The linking of custom handlers is based on the Nodes' type definitions. In other words, nodes can have handlers assigned to write and read requests based on their type. In practice, this happens when a HasTypeDefinition reference is added to a Node. The CustomObjectManager checks if a custom handler has been registered for the type, and if one has, it places a reference to it in the Node. Upon reading or writing of the Node, the custom code will be called.

The handlers must be hard coded into CustomNodeManager in this implementation, because use cases which require new handlers (meaning custom code) to be dynamically added are likely to be very rare. However, if necessary, Methods for adding new code to the CustomNodeManager could be added into the server's AddressSpace, allowing for remote addition of code. This kind of solution would be necessary if a configuration client was used to add code.

A callback interface was also added to CustomObjectManager that allows handlers to access other Nodes' information. This is the kind of solution as was described in section 5.4 as 'Code running beside the Nodes'. The callback interface is called NodeObjectAccess and it has two methods. getNode() allows handlers to request other Nodes from the CustomObjectManager, so that their data may be accessed. Nodes in other NodeManagers may not be accessed this way. convertToLocalId() allows the handler to convert ExpandedNodeIds to NodeIds according to the server's namespace table. This method is necessary as the target Node of References is referred to by ExpandedNodeId and needs to be converted to NodeId before requesting the Node. NodeObjectAccess is presented in Appendix B.

## 6.5 Example use case

This section presents a simple use case in which the developed solution has been used.

### 6.5.1 Overview

The idea in this example use of the developed system is to load a set of Nodes at startup to the server's Address Space and use a custom code binding to change the behavior of Nodes upon reading and writing. The example is a simple calculation to determine the average of some values. The average may be read as the value of a Variable, and the values from which it is calculated are visible in the AddressSpace as well.

### 6.5.2 Definition of the example

To create the example on the server three things are needed: an XML which defines the Nodes that are to be added to the server, the custom code that will be linked to the Node, and defining the CustomObjectManager to link the Node to the handler containing the custom code. The XML definition (conforming to UA Model Design.xsd) of the example Nodes is presented below.

```
<?xml version="1.0" encoding="utf-8"?>
<opc:ModelDesign xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:opc="http://opcfoundation.org/UA/ModelDesign.xsd"
                 xmlns:ua="http://opcfoundation.org/UA/"
                 xmlns:uax="http://opcfoundation.org/UA/2008/02/Types.xsd"
                 xmlns="http://prosysopc.com/UAJava/TestNamespace"
                 TargetNamespace="http://prosysopc.com/UAJava/TestNamespace">
  <opc:Namespaces>
    <opc:Namespace Name="OpcUa"
                   XmlNamespace="http://opcfoundation.org/UA/2008/02/Types.xsd"
                   Prefix="Opc.Ua"
                   InternalPrefix="Opc.Ua.Server">http://opcfoundation.org/UA/</opc:Namespace>
    <opc:Namespace Name="ProsysJavaTest"
                   Prefix="Prosys">http://prosysopc.com/UAJava/TestNamespace</opc:Namespace>
  </opc:Namespaces>
  <opc:Object SymbolicName="Examples" SupportsEvents="true">
    <opc:Children>
      <opc:Variable SymbolicName="Average" SupportsEvents="true">
        <opc:Children>
          <opc:Property SymbolicName="A" DataType="ua:Double" DefaultValue="2"/>
          <opc:Property SymbolicName="B" DataType="ua:Double" DefaultValue="3"/>
          <opc:Property SymbolicName="C" DataType="ua:Double" DefaultValue="5"/>
        </opc:Children>
        <opc:References>
          <opc:Reference>
            <opc:ReferenceType>ua:HasTypeDefinition</opc:ReferenceType>
            <opc:TargetId>AverageType</opc:TargetId>
          </opc:Reference>
        </opc:References>
      </opc:Variable>
    </opc:Children>
  </opc:Object>
  <opc:VariableType SymbolicName="AverageType" />

</opc:ModelDesign>
```

First there is the XML header and definitions for namespaces. After those an Object, 'Examples' is defined as a folder. It has a child Variable, 'Average', which in turn has three Properties, 'A', 'B' and 'C'. They are all of type ua:Double and have default values of 2, 3 and 5. 'Average' is also defined to have a type definition, AverageType. The AverageType is declared next in the XML. Since Examples does not have any explicit parent, it will be placed as a child of Objects (a top level folder for Nodes of type Object), and similarly AverageType will be a child of BaseVariableType (a superclass for all VariableTypes). It is important to note that Average's type is defined here, since it is what will be used to bind the custom functionality of calculating averages to it later.

In addition to the XML, a .csv file is necessary to specify the numeric NodeIds for the Nodes. It has a very simple format, the SymbolicName of each Node, followed by its numeric id and NodeClass. The contents of the file is presented below.

```
Examples,1,Object
Examples_Average,11,Variable
Examples_Average_A,12,Variable
Examples_Average_B,13,Variable
Examples_Average_C,14,Variable
AverageType,1001,VariableType
```

ExampleDataSource is a very simple custom data source. In this example, its purpose is to read the values of the read Node's properties and to calculate an average of them. Below the code of ExampleDataSource, and thus a model of what a handler looks like at its simplest, is presented. Note that the Node cannot be written to, and if a client attempts this, a statuscode signalling this is sent back to the client.

```
package fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.infomodel.example;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;

import org.opcfoundation.ua.builtintypes.*;
import org.opcfoundation.ua.common.ServiceResultException;
import org.opcfoundation.ua.core.Identifiers;
import org.opcfoundation.ua.core.Node;
import org.opcfoundation.ua.core.ReferenceNode;
import org.opcfoundation.ua.core.VariableNode;
import org.opcfoundation.ua.utils.NumericRange;

import fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.nodemanager.IDataSource;
import fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.session.IIdentityContext;
import fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.session.Session;

public class ExampleDataSource implements IDataSource {

  private NodeObjectAccess customManager;
  private static ExampleDataSource instance = null;

  private ExampleDataSource(NodeObjectAccess c){
    this.customManager = c;
  }

  public static ExampleDataSource getInstance(NodeObjectAccess c) throws Exception{
    if(instance == null){
      ExampleDataSource.instance = new ExampleDataSource(c);
      return instance;
    } else {
      if(!instance.customManager.equals(c))
        throw new Exception();
      return instance;
    }
  }

  @Override
  public ServiceResult read(IIdentityContext context, NodeId nodeId,
      Object handle, UnsignedInteger attributeId, String range,
      DataValue value, Calendar sourceTimestamp)
      throws ServiceResultException {
    // Fetching the Properties of the Node
    ReferenceNode[] references = ((Node)handle).getReferences();
    List<VariableNode> propertyNodes = new ArrayList<VariableNode>();
    for(ReferenceNode r : references){
      if(r.getReferenceTypeId().equals(Identifiers.HasProperty)){
        // Converting ExpandedNodeId to NodeId
        NodeId propertyId = customManager.convertToLocalId(r.getTargetId());
        // Fetching the Node, which is a Variable, and adding it to the list
        VariableNode propertyNode = (VariableNode)customManager.getNode(propertyId);
        propertyNodes.add(propertyNode);
      }
    }

    // Calculating the average
    double sum = 0;
    double count = 0;
    for(VariableNode v : propertyNodes){
      sum += (Double)v.getValue().getValue();
      count++;
    }
    double average = sum/count;

    // Returning the result
    value.setValue(new Variant(average));
    return new ServiceResult(StatusCode.GOOD);
```

```
    }

    @Override
    public ServiceResult write(Session session, NodeId nodeId, Object handle,
        UnsignedInteger attributeId, NumericRange range, DataValue value) {
      return new ServiceResult(StatusCode.BAD);
    }

}
```

The code binding is done in the CustomObjectManager by placing an instance of ExampleDataSource as a handler for all Nodes that have their HasTypeDefinition pointing to AverageType. When such a reference is added, the handler is also placed in the Node and subsequentially called upon reads and writes. This way, all Nodes of type AverageType will have the same functionality when read or written to.

### 6.5.3 Control flow and execution

This section presents the control flow for reading the value of Average. At server startup, the AverageType and the Average instance are loaded from an XML and placed into the server using the mechanisms described above. In addition, a special handler, ExampleDataSource, has been defined for read calls for Variables of type AverageType. The CustomObjectManager, which has registered itself as the NodeManager for the namespace Average is going to be placed in, will add the handler to Average when the HasTypeDefinition Reference is added from Average to AverageType, or in other words, when Average is defined to be of AverageType. After this, all read calls to Average will end up in ExampleDataSource. The ExampleDataSource will request Average's properties from CustomObjectManager, and determine the value to be returned as the average of the properties' values. The value is then returned to the client. Figure 24 presents the read as a sequence diagram.

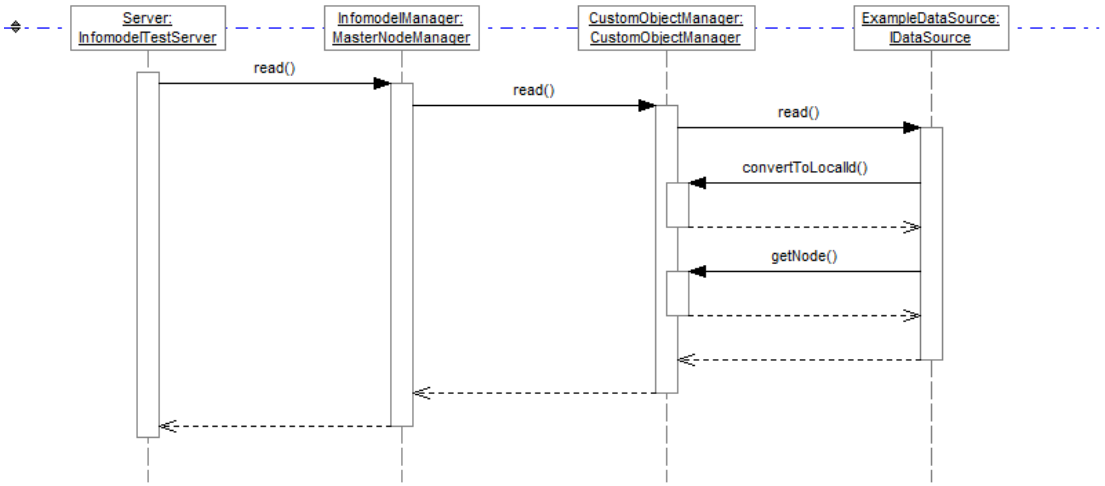


Figure 24: A sequence diagram showing a read call handled by ExampleDataSource.

After the server has started up, the Nodes are accessible in its AddressSpace. Figure

25 shows the example Nodes viewed in an OPC UA Client (UAExpert by Unified Automation[39]). The properties A, B and C can be seen in the AddressSpace.

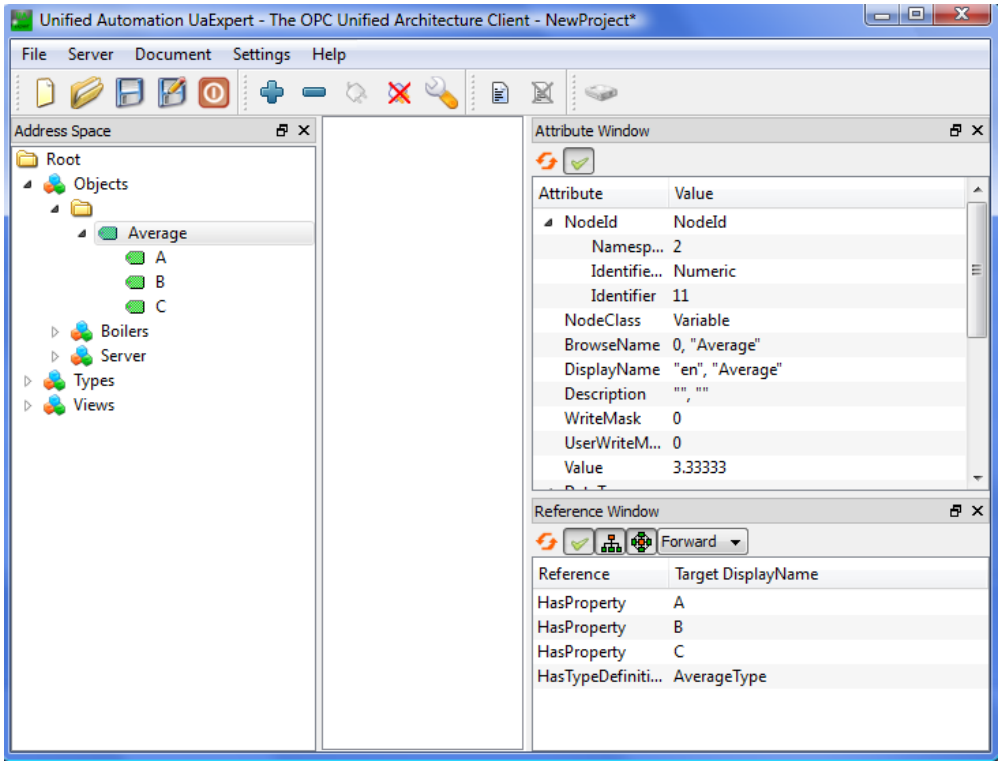


Figure 25: The example use case viewed in an OPC UA Client.

This example demonstrates that instantiation of Nodes and Types into a server's AddressSpace can be done without code generation, and that such a solution can also include custom code. The example is instantiated at startup, but if any Nodes of the type AverageType were added to the correct namespace during runtime, the exact same binding would be applied to them and the handler added.

# 7 Conclusions

This thesis has examined the requirements for implementing information model support in OPC UA servers. OPC UA provides good tools for utilizing and handling the semantics of data. Whereas data was flat in OPC, in UA it is modelled as a full mesh of interconnected nodes, which convey information model of the data. Information models are a necessary component of more intelligent applications and taking advantage of semantic data. OPC UA servers must provide this information to clients if it is to be used meaningfully. Different use cases can be determined for both design-time and runtime situations. Depending on the demands, different implementations must be used to satisfy the requirements posed by the use cases.

Information models may be defined based on the base OPC UA Address Space Model, or using one of many companion specifications as base. The companion specifications are made by various organizations and are meant to harmonize the way information is represented in a specific application field, thus furthering interoperability. Companion specifications have already been released, and more are under development work. They will serve as base for many OPC UA information models, and are a vital part of using information models in applications to their fullest potential. Information models may be represented by graphs or definition files, and can be designed both by hand and by designer software. Software products that can be used for information model design work are already on the market and more will be released as the popularity of OPC UA rises.

An implementation has been made as part of the Java Sample Server. The server now supports direct instantiation of information models from compliant XML files as well as binding data with custom code. Handling information models dynamically at runtime requires a more complex implementation in the server as well as an effective way to perform possible data bindings to underlying data sources. In this case, the XML format used in the .NET reference implementation was used, and a framework for its use developed as part of the Java Sample Server. The models which are taken into use in the .NET implementation by using code generation are dynamically added to the Java Sample Server without code generation, enabling changes to the information models while the server is running. The server also supports a simple system of binding custom data to the server's nodes.

An example of instantiating an information model from an XML file to the Java Sample Server's AddressSpace as well as binding custom functionality to a node's value was demonstrated. While the example is very simple, it serves as a proof-of-concept for the idea of implementing a server's address space without generating any code, while retaining the ability to have custom code in nodes. Several other use cases for code binding exist that are not covered by this thesis, and a fully featured implementation that covers all of them would require more research and development.

The objectives laid out in the beginning of this thesis have been met for the most part. Java Sample Server has support for information models and related service

calls from clients, and it also supports binding the server to underlying data sources, i.e. custom code. Several implementation choices were considered, and the positive and negative sides of each were studied. The question how information models should be used in applications was not deeply covered, unlike originally planned. As this is more related to the client side applications, it was not delved in as thoroughly as originally thought. Overall, the objectives were fulfilled and the results can be used in further development. The implementation presented in this thesis will serve as a base to the information model implementation in the Prosys OPC UA Java SDK Server.

OPC UA is a powerful specification that will no doubt be the next industry standard. The hold of OPC is very strong, however, and it will take time before UA gains widespread popularity in industrial installations. Much of the appeal of OPC UA is due to the new and inbuilt possibility to define information models and use the conveyed semantics in applications. While the specifications now permit such applications, the best uses will take time to be discovered. Even while it is easy to see benefits in information modelling now, the true power of information models will only be seen in the future.

# References

[1] Antoniou, Grigoris and van Harmelen, Frank. *A Semantic Web Primer*. 2nd edition. MIT Press, 2008.

[2] Damm, Matthias, Leitner, Stefan-Helmut and Mahnke, Wolfgang. *OPC Unified Architecture*. Springer, 2009.

[3] OPC Foundation. OPC UA SDK 1.01 Source Code. 2009. http://www.opcfoundation.org/

[4] OPC Foundation. OPC Overview 1.00. 1998. http://www.opcfoundation.org/

[5] Hannelius et al. Roadmap to Adopting OPC UA. Industrial Informatics, July 2008, pp. 756-761. DOI:10.1109/INDIN.2008.4618203. 2008.

[6] OPC Foundation. OPC Unified Architecture Specification, Part 1: Overview and Concepts. Release 1.01. 2009.

[7] OPC Foundation. OPC Unified Architecture Specification, Part 3: Address Space Model. Release 1.01. 2009.

[8] OPC Foundation. OPC Unified Architecture Specification, Part 4: Services. Release 1.01. 2009.

[9] OPC Foundation. OPC Unified Architecture Specification, Part 5: Information Model. Release 1.01. 2009.

[10] OPC Foundation. OPC Unified Architecture Specification, Part 6: Mappings. Release 1.00. 2009.

[11] OPC Foundation. OPC Unified Architecture Draft Specification, Devices. Draft Version 0.75. 2008.

[12] OPC Foundation. OPC Unified Architecture Draft Specification, Analyser Devices. Draft Version 0.31. 2009.

[13] OPC Foundation. OPC UA For IEC 61131-3 RC 0.09.1 Companion Specification. Release Candidate 0.09.1.

[14] Hannelius et al. Embedding OPC Unified Architecture. 2009. Available at http://www.wapice.com/wapice_cms/files/HanneliusSchroffTuominen_%20Embedding_OPCUA.pdf (13.1.2010)

[15] Luth, Jim. OPC-UA Architecture. Presentation slides, OPC Unified Architecture DevCon, 2007.

[16] Mahnke, Wolfgang. Information Model and Services. Presentation slides, OPC Unified Architecture DevCon, 2007.

[17] Leitner, Stefan-Helmut and Mahnke, Wolfgang. OPC UA - Service-oriented Architecture for Industrial Applications. Available at http://cimug.ucaiug.mobi/KB/Knowledge%20Base/SOA%20for%20Industrial%20Applications.pdf (13.1.2010)

[18] Brandl, Dennis. OPC and MES. Presentation and slides at OPC Day 2009.

[19] Prosys PMS Ltd. OPC UA Client Java SDK & OPC UA Server Java SDK. 2009. http://www.prosysopc.com/

[20] Unified Automation GmbH. OPC UA Server Development Kit. 2009. http://www.unified-automation.com/server-sdk.htm

[21] Damm, Matthias. Introduction UA Server SDK. Lecture slides, OPC UA Developer's Workshop. 2009.

[22] Nykänen, Matti. Object-oriented information model for a variable speed drive. Master's Thesis, Helsinki University of Technology, Department of Automation and Systems Technology, Espoo, 2008.

[23] Neste Jacobs. NAPCON - Process Control Technology. 2009. http://www.nestejacobs.com/

[24] Aro, Jouni and Frejborg, Andreas. Adding OPC UA Server and Client with .NET to NAPCON Process Control Database. Presentation and slides at OPC Day 2009.

[25] Frejborg, Andreas. Personal communication on 12.1.2010.

[26] Aarnio et al. Architectural Perspectives for Factory Information Systems In Manufacturing Operations and Control. ProductionPRO Report. 2008.

[27] IDS-ADI. RDS/WIP 1.0. http://rdl.rdlfacade.org/

[28] Gulla et al. Semantic Interoperability in the Norwegian Petroleum Industry. Lecture Notes in Informatics. 2006. Available at http://subs.emis.de/LNI/Proceedings/Proceedings84/GI-Proceedings-84-6.pdf (13.1.2010)

[29] Damm, Matthias. Personal communication on 14.08.2009.

[30] Unified Automation GmbH. Technical Preview for UaModeler - Windows. http://www.unified-automation.com/

[31] CAS. OPC UA Address Space Model Designer. 2009. http://www.commsvr.com/Products/UAModelDesigner.aspx

[32] VTT. Simantics. https://www.simantics.org/

[33] XStream library. 2008. http://xstream.codehaus.org/

[34] Armstrong, Randy. Personal communication 13.8.2009.

[35] Harold, Elliotte Rusty. *Processing XML with Java*. Addison Wesley 2006.

[36] The Castor Project. http://www.castor.org/

[37] jaxb: JAXB Reference Implementation. 2009. https://jaxb.dev.java.net/

[38] Dennis M. Sosnoski. JiBX: Binding XML to Java Code. 2009. http://jibx.sourceforge.net/

[39] Unified Automation GmbH. UaExpert Technical Preview 1.0.0. 2009. http://www.unified-automation.com/

# Appendix A - UAModelDesign.xsd schema

This appendix the XML Schema that has been used in this thesis, UA Model Design.xml. It is directly copied from the OPC Foundation .NET SDK[3], but has been reformatted for this thesis.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema
  targetNamespace="http://opcfoundation.org/UA/ModelDesign.xsd"
  elementFormDefault="qualified"
  xmlns="http://opcfoundation.org/UA/ModelDesign.xsd"
  xmlns:mstns="http://opcfoundation.org/UA/ModelDesign.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:html="http://www.w3.org/1999/xhtml"
>
  <xs:element name="ModelDesign">
    <xs:annotation>
      <xs:documentation>
        <html:p>
        The root element for the information model.<html:br />

        This scheme allows information modellers to defined UA type in a machine readable
        form. This definition can be used to generate code and documentation.<html:br />

        The file is expected to contain a number of types and their instance declarations.
        Objects which are unique in the address space can also be defined.<html:br />

        A validator is available verify consistancy of the model generator and to create
        suitable values for optional information. Once the design is validated it can be
        passed to a generator which creates different types of code or documentation.<html:br />
        </html:p>
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Namespaces" type="NamespaceTable" minOccurs="0" />
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="ObjectType" type="ObjectTypeDesign" minOccurs="0" />
          <xs:element name="VariableType" type="VariableTypeDesign" minOccurs="0" />
          <xs:element name="ReferenceType" type="ReferenceTypeDesign" minOccurs="0" />
          <xs:element name="DataType" type="DataTypeDesign" minOccurs="0" />
          <xs:element name="Method" type="MethodDesign" minOccurs="0" />
          <xs:element name="Object" type="ObjectDesign" minOccurs="0" />
          <xs:element name="Variable" type="VariableDesign" minOccurs="0" />
          <xs:element name="Property" type="PropertyDesign" minOccurs="0" />
          <xs:element name="Dictionary" type="DictionaryDesign" minOccurs="0" />
          <xs:element name="View" type="ViewDesign" minOccurs="0" />
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="TargetNamespace" type="xs:string" use="required" />
      <xs:attribute name="TargetXmlNamespace" type="xs:string" use="optional" />
      <xs:attribute name="DefaultLocale" type="xs:string" use="optional" default="en" />
      <xs:anyAttribute processContents="lax" />
    </xs:complexType>
  </xs:element>

  <xs:complexType name="NamespaceTable">
    <xs:annotation>
      <xs:documentation>
        <html:p>
        This defines the namespaces used in the model.<html:br />

        Each namespace listed should also have a namespace prefix defined in the xs:schema
        element.<html:br />

        The order of the namespaces is significant and used to assigned a numeric index to
        namespaces when they are used in BrowsePaths specified in the ModelDesign.
        </html:p>
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="Namespace" type="Namespace" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Namespace">
    <xs:annotation>
      <xs:documentation>
        <html:p>
          Defines a single namespace along with identifiers for the namespace.<html:br />

          The Name is used to create a progam constant for the URL.<html:br />

          The Prefix is the C# namespace which qualifies the generated types.<html:br />

          The InternalPrefix is an optional C# namespace which qualifies the generated
          types used only by the server.<html:br />
        </html:p>
```

```
    </xs:documentation>
  </xs:annotation>
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="Name" type="xs:string" use="optional">
        <xs:annotation>
          <xs:documentation>
            A symbolic name for the namespace that can used as a variable name.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="Prefix" type="xs:string" use="optional">
        <xs:annotation>
          <xs:documentation>
            The .NET namespace used for the classes produced by the generator.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="InternalPrefix" type="xs:string" use="optional">
        <xs:annotation>
          <xs:documentation>
            The .NET namespace used for classes that are only used within a server
            application.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="XmlNamespace" type="xs:string" use="optional">
        <xs:annotation>
          <xs:documentation>
            The URI for the XML namespace which the data types belong to if it is
            different from the URI for the model namespace.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="XmlPrefix" type="xs:string" use="optional">
        <xs:annotation>
          <xs:documentation>
            The prefix to be used in the XML file for the XML namespace which the
            data types belong to. Used only XmlNamespace is set.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="FilePath" type="xs:string" use="optional">
        <xs:annotation>
          <xs:documentation>
            The path to the file containing the design file for the namespace.
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="NodeDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        The base type of all node designs.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="BrowseName" type="xs:string" minOccurs="0">
      <xs:annotation>
        <xs:documentation>
          <html:p>
            The BrowseName is the name used in the information model. The validator
            will create the BrowseName automatically from the SymbolicName. The
            BrowseName is qualified by the namespace used for the SymbolicName.
          </html:p>
        </xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="DisplayName" type="LocalizedText" minOccurs="0">
      <xs:annotation>
        <xs:documentation>
          <html:p>
            The DisplayName human readable name for the Node. This element includes
            an optional key that can be used to look up the display name for other
            locales in a resource DB. The validator automatically creates the
            DisplayName from the BrowseName.
          </html:p>
        </xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="Description" type="LocalizedText" minOccurs="0">
      <xs:annotation>
        <xs:documentation>
          <html:p>
            The Description the value of the Description attribute for the Node.
            This element includes an optional key that can be used to look up the
            Description for other locales in a resource DB. The validator automatically
```

```
          creates a generic Description from the BrowseName and NodeClass.
        </html:p>
      </xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="Children" type="ListOfChildren" minOccurs="0">
    <xs:annotation>
      <xs:documentation>
        <html:p>
          The Children are the Properties or Components of a Node.
        </html:p>
      </xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="References" type="ListOfReferences" minOccurs="0">
    <xs:annotation>
      <xs:documentation>
        <html:p>
          The References specify additional references from the Node. These
          references may refer to other children of the same Node or children
          of other Nodes defined in the ModelDesign.
        </html:p>
      </xs:documentation>
    </xs:annotation>
  </xs:element>
</xs:sequence>
<xs:attribute name="SymbolicName" type="xs:QName" use="optional">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        The SymbolicName identifies the Node within the the ModelDesign or within
        the containing Node. The SymbolicName should always be specified. It is
        used to create the BrowseName and SymbolicId if they are not specified.
      </html:p>
    </xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="SymbolicId" type="xs:QName" use="optional">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        The SymbolicId is a globally unique identifier for the Node. The validator
        will create the SymbolicId automatically from the SymbolicName if it is not
        specified.
      </html:p>
    </xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="IsDeclaration" type="xs:boolean" use="optional" default="false">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        The IsDeclaration flag indicates that the Node is defined elsewhere and no
        code will be generated. Nodes that are declarations do not need to be
        completely defined. They only need to have the information required to
        generate code for nodes that reference it (e.g. the BaseType).
      </html:p>
    </xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="NumericId" type="xs:unsignedInt" use="optional">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        The NumericId specifies the unique numeric id for the Node. It is filled
        in automatically by reading a CSV file containing the SymbolicIds and an
        associated UInt32. The validator will automatically assign a unique id if
        no CSV input is provided.<html:br />

        The NumericId or StringId are combined with the Namespace used for the
        SymbolicId to create the well known UA NodeId for the Node. The generator
        will create programmatic constants that can be used to reference the Node
        in code.
      </html:p>
    </xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="StringId" type="xs:string" use="optional">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        The StringId is an alternate unique identfier for the node. It is used
        instead of the NumericId if it is specified in the CSV input file.
      </html:p>
    </xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="WriteAccess" type="xs:unsignedInt" use="optional" default="0">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        The bit mask which indicates which attributes are writeable.
```

```
        </html:p>
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:complexType>

<xs:complexType name="ViewDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        A View Node (not supported by the validator at this time).
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="NodeDesign">
      <xs:attribute name="SupportsEvents" type="xs:boolean" use="optional" default="false">
        <xs:annotation>
          <xs:documentation>
            <html:p>
              Whether the View generates events.
            </html:p>
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="ContainsNoLoops" type="xs:boolean" use="optional" default="false">
        <xs:annotation>
          <xs:documentation>
            <html:p>
              Specifies that the View contains a non-looping hierarchy.
            </html:p>
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="TypeDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        A base type for all Type Nodes (ObjectType, VariableType, DataType and ReferenceType).
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="NodeDesign">
      <xs:sequence>
        <xs:element name="ClassName" type="xs:string" minOccurs="0">
          <xs:annotation>
            <xs:documentation>
              <html:p>
                This is the name for the instance of the type. If not specified the validator
                creates it by removing the 'Type' suffix from the SymbolicName for the Node.
              </html:p>
            </xs:documentation>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="BaseType" type="xs:QName" use="optional">
        <xs:annotation>
          <xs:documentation>
            <html:p>
              The SymbolicId for the BaseType.
            </html:p>
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="IsAbstract" type="xs:boolean" use="optional" default="false">
        <xs:annotation>
          <xs:documentation>
            <html:p>
              Whether the Type is abstract.
            </html:p>
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="NoClassGeneration" type="xs:boolean" use="optional" default="false">
        <xs:annotation>
          <xs:documentation>
            <html:p>
              Whether to supress class generation for the type.
            </html:p>
          </xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="ObjectTypeDesign">
```

```
  <xs:annotation>
    <xs:documentation>
      <html:p>
        ObjectTypes define structure of an Object in the information model.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="TypeDesign">
      <xs:attribute name="SupportsEvents" type="xs:boolean" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="VariableTypeDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        VariableTypes define structure of a Variable in the information model.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="TypeDesign">
      <xs:sequence>
        <xs:element name="DefaultValue" type="DefaultValue" minOccurs="0" />
      </xs:sequence>
      <xs:attribute name="DataType" type="xs:QName" use="optional" />
      <xs:attribute name="ValueRank" type="ValueRank" use="optional" />
      <xs:attribute name="ArrayDimensions" type="xs:string" use="optional" />
      <xs:attribute name="AccessLevel" type="AccessLevel" use="optional" />
      <xs:attribute name="MinimumSamplingInterval" type="xs:int" use="optional" />
      <xs:attribute name="Historizing" type="xs:boolean" use="optional" />
      <xs:attribute name="ExposesItsChildren" type="xs:boolean" use="optional" default="false" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="DataTypeDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        DataTypes define structure of a Value for Variables in the information model.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="TypeDesign">
      <xs:sequence>
        <xs:element name="Fields" type="ListOfFields" minOccurs="0" />
        <xs:element name="Encodings" type="ListOfEncodings" minOccurs="0" />
      </xs:sequence>
      <xs:attribute name="NoArraysAllowed" type="xs:boolean" use="optional" default="false" />
      <xs:attribute name="NotInAddressSpace" type="xs:boolean" use="optional" default="false" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="ReferenceTypeDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        ReferenceType define typed references between Nodes.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="TypeDesign">
      <xs:sequence>
        <xs:element name="InverseName" type="LocalizedText" minOccurs="0" />
      </xs:sequence>
      <xs:attribute name="Symmetric" type="xs:boolean" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="InstanceDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        A base type for all Instance Nodes (Object, Variable, and Method).
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="NodeDesign">
      <xs:sequence>
        <xs:element name="ReferenceType" type="xs:QName" minOccurs="0" />
      </xs:sequence>
      <xs:attribute name="Declaration" type="xs:QName" use="optional" />
      <xs:attribute name="TypeDefinition" type="xs:QName" use="optional" />
      <xs:attribute name="ModellingRule" type="ModellingRule" use="optional" />
```

```
      <xs:attribute name="MinCardinality" type="xs:unsignedInt" use="optional" default="0" />
      <xs:attribute name="MaxCardinality" type="xs:unsignedInt" use="optional" default="0" />
      <xs:attribute name="PreserveDefaultAttributes" type="xs:boolean" use="optional" default="false" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="ObjectDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        Defines the structure of an Object in the information model.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="InstanceDesign">
      <xs:attribute name="SupportsEvents" type="xs:boolean" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="VariableDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        Defines the structure of a Variable in the information model.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="InstanceDesign">
      <xs:sequence>
        <xs:element name="DefaultValue" type="DefaultValue" minOccurs="0" />
      </xs:sequence>
      <xs:attribute name="DataType" type="xs:QName" use="optional" />
      <xs:attribute name="ValueRank" type="ValueRank" use="optional" />
      <xs:attribute name="ArrayDimensions" type="xs:string" use="optional" />
      <xs:attribute name="AccessLevel" type="AccessLevel" use="optional" />
      <xs:attribute name="MinimumSamplingInterval" type="xs:int" use="optional" />
      <xs:attribute name="Historizing" type="xs:boolean" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="MethodDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        Defines the a Method in the information model.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="InstanceDesign">
      <xs:sequence>
        <xs:element name="InputArguments" type="ListOfArguments" minOccurs="0" />
        <xs:element name="OutputArguments" type="ListOfArguments" minOccurs="0" />
      </xs:sequence>
      <xs:attribute name="NonExecutable" type="xs:boolean" use="optional" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="PropertyDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        Defines a Variable which is a Property for a Node.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="VariableDesign">
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="EncodingDesign">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        Defines an Object which is a DataTypeEncoding for a DataType.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="ObjectDesign" />
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="DictionaryDesign">
```

```
  <xs:annotation>
    <xs:documentation>
      <html:p>
        Defines an Variable which is a DataTypeDictionary.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent mixed="false">
    <xs:extension base="VariableDesign">
      <xs:attribute name="EncodingName" type="xs:QName" use="required" />
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="Reference">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        Defines a reference between two nodes.<html:br />

        The SourceId is the SymbolicId of the Node that contains the Reference.<html:br />

        The SourcePath and TargetPath are RelativePaths specified using the syntax
        defined in Part 4. The order of the Namespaces defined in the Namespaces element
        is used to determine the namespace index used in the RelativePaths.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="ReferenceType" type="xs:QName" minOccurs="1" />
    <xs:element name="TargetId" type="xs:QName" minOccurs="1" />
  </xs:sequence>
  <xs:attribute name="IsInverse" type="xs:boolean" use="optional" default="false" />
  <xs:attribute name="IsOneWay" type="xs:boolean" use="optional" default="false" />
</xs:complexType>

<xs:complexType name="ListOfReferences">
  <xs:sequence>
    <xs:element name="Reference" type="Reference" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Parameter">
  <xs:annotation>
    <xs:documentation>
      <html:p>
        Defines a Field in a DataType or Argument of a Method.
      </html:p>
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="Description" type="LocalizedText" minOccurs="0" />
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="optional" />
  <xs:attribute name="Identifier" type="xs:int" use="optional" />
  <xs:attribute name="DataType" type="xs:QName" use="optional" />
  <xs:attribute name="ValueRank" type="ValueRank" use="optional" default="Scalar" />
</xs:complexType>

<xs:complexType name="ListOfArguments">
  <xs:sequence>
    <xs:element name="Argument" type="Parameter" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="ListOfFields">
  <xs:sequence>
    <xs:element name="Field" type="Parameter" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="ListOfEncodings">
  <xs:sequence>
    <xs:element name="Encoding" type="EncodingDesign" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="ListOfChildren">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="Object" type="ObjectDesign" minOccurs="0" />
      <xs:element name="Variable" type="VariableDesign" minOccurs="0" />
      <xs:element name="Property" type="PropertyDesign" minOccurs="0" />
      <xs:element name="Method" type="MethodDesign" minOccurs="0" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="LocalizedText">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="Key" type="xs:string" use="optional" default="" />
```

```
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:complexType name="DefaultValue">
    <xs:sequence>
      <xs:any minOccurs="0" processContents="lax"/>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="AccessLevel">
    <xs:restriction base="xs:string">
      <xs:enumeration value="None" />
      <xs:enumeration value="Read" />
      <xs:enumeration value="Write" />
      <xs:enumeration value="ReadWrite" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="ValueRank">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Scalar" />
      <xs:enumeration value="Array" />
      <xs:enumeration value="ScalarOrArray" />
      <xs:enumeration value="OneOrMoreDimensions" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="ModellingRule">
    <xs:restriction base="xs:string">
      <xs:enumeration value="None" />
      <xs:enumeration value="Mandatory" />
      <xs:enumeration value="Optional" />
      <xs:enumeration value="ExposesItsArray" />
      <xs:enumeration value="CardinalityRestriction" />
      <xs:enumeration value="MandatoryShared" />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

# Appendix B - Interface definitions

This appendix presents the interfaces used in the solution.

AddressSpaceHandling is AddressSpaceHandler's interface for ModelParser and the server.

```
package fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.infomodel.interfaces;

import org.opcfoundation.ua.builtintypes.DiagnosticInfo;
import org.opcfoundation.ua.builtintypes.StatusCode;
import org.opcfoundation.ua.common.ServiceFaultException;
import org.opcfoundation.ua.core.AddNodesItem;
import org.opcfoundation.ua.core.AddNodesResult;
import org.opcfoundation.ua.core.AddReferencesItem;
import org.opcfoundation.ua.core.DeleteNodesItem;
import org.opcfoundation.ua.core.DeleteReferencesItem;
import fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.infomodel.ChildRelation;
import fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.nodemanager.INodeManager;

public interface AddressSpaceHandling {

    // Methods from the NodeManagerServiceSet
    void addNodes(AddNodesItem[] nodesToAdd,
                AddNodesResult[] results,
                DiagnosticInfo[] diagnosticInfos) throws ServiceFaultException;

    void addReferences(AddReferencesItem[] referencesToAdd,
                StatusCode[] results,
                DiagnosticInfo[] diagnosticInfos) throws ServiceFaultException;

    void deleteNodes(DeleteNodesItem[] nodesToDelete,
                StatusCode[] results,
                DiagnosticInfo[] diagnosticInfos) throws ServiceFaultException;

    void deleteReferences(DeleteReferencesItem[] referencesToDelete,
                StatusCode[] results,
                DiagnosticInfo[] diagnosticInfos) throws ServiceFaultException;


    void registerNamespaceManager(INodeManager namespaceManager,
                String uri) throws Exception;

    void registerChildren(ChildRelation[] relations);

}
```

NodeObjectAccess is a callback interface that is implemented by CustomObject-Manager and meant to be used by data sources.

```
package fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.infomodel.example;

import org.opcfoundation.ua.builtintypes.ExpandedNodeId;
import org.opcfoundation.ua.builtintypes.NodeId;
import org.opcfoundation.ua.core.Node;

public interface NodeObjectAccess {

    /* Method for DataSources to access other nodes in the NodeManager's
     * address space
     */
    public Node getNode(NodeId nodeId);

    // Method for converting ExpandedNodeIds into local NodeIds
    public NodeId convertToLocalId(ExpandedNodeId targetId);

}
```

NodeManagement and ReferenceManagement may or may not be implemented by custom NodeManagers. Together they represent the same services as the whole NodeManagementServiceSet.

```
package fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.infomodel.interfaces;

import org.opcfoundation.ua.builtintypes.DiagnosticInfo;
import org.opcfoundation.ua.builtintypes.StatusCode;
import org.opcfoundation.ua.common.ServiceFaultException;
import org.opcfoundation.ua.core.AddNodesItem;
import org.opcfoundation.ua.core.AddNodesResult;
import org.opcfoundation.ua.core.DeleteNodesItem;

public interface NodeManagement {

    void addNodes(AddNodesItem[] nodesToAdd,
                AddNodesResult[] results,
                DiagnosticInfo[] diagnosticInfos) throws ServiceFaultException;

    void deleteNodes(DeleteNodesItem[] nodesToDelete,
                StatusCode[] results,
                DiagnosticInfo[] diagnosticInfos) throws ServiceFaultException;

}
```

```
package fi.uajavaproject.opcuatoolkitNEW.uaserverlibrary.infomodel.interfaces;

import org.opcfoundation.ua.builtintypes.DiagnosticInfo;
import org.opcfoundation.ua.builtintypes.StatusCode;
import org.opcfoundation.ua.common.ServiceFaultException;
import org.opcfoundation.ua.core.AddReferencesItem;
import org.opcfoundation.ua.core.DeleteReferencesItem;

public interface ReferenceManagement {

    void addReferences(AddReferencesItem[] referencesToAdd,
                StatusCode[] results,
                DiagnosticInfo[] diagnosticInfos) throws ServiceFaultException;

    void deleteReferences(DeleteReferencesItem[] referencesToDelete,
                StatusCode[] results,
                DiagnosticInfo[] diagnosticInfos) throws ServiceFaultException;

}
```